Anais da
II Oficina Nacional em
Problemas Combinatórios:
Teoria, Algoritmos e Aplicações

Marcus Vinicius S. Poggi de Aragão
Cid Carvalho de Souza
(*editores*)

Relatório Técnico DCC–95-17

Novembro de 1995

Anais da
II Oficina Nacional em
Problemas Combinatórios:
Teoria, Algoritmos e Aplicações

Marcus Vinicius S. Poggi de Aragão
Cid Carvalho de Souza
(*editores*)

DCC-IMECC-UNICAMP
15 a 17 de Novembro de 1995

Abstract: This report contains the Proceedings of the II National Workshop on Combinatorial Problems: Theory, Algorithms and Applications (II ON ProComb). The workshop was held November 15–17, 1995 at the Computer Science Department (DCC-IMECC) of the State University of Campinas (UNICAMP), Campinas, SP, Brazil.

The event was part of the ProComb project, comprising combinatorics researchers from USP, UNICAMP, PUC-RJ and UFRJ. The project is sponsored by CNPq though its Multi-institutional Thematic Program in Computer Science (ProTeM-CC-II).

Sumário: Este relatório contém os anais da segunda Oficina Nacional em Problemas Combinatórios: Teoria, Algoritmos e Aplicações. A oficina foi realizada de 15 a 17 de novembro de 1995, no Depto. de Ciência da Computação (DCC-IMECC) da UNICAMP.

O evento fez parte do projeto "ProComb" que reuniu pesquisadores da USP, UNICAMP, PUC-RJ e UFRJ que atuam na área de combinatória. Este projeto foi financiado pelo CNPq dentro do Programa Temático Multi-institucional em Ciência da Computação, ProTeM-CC-II.

ProComb project members:

Arnaldo Mandel (USP)
Carlos Eduardo Ferreira (USP)
Célia P. de Mello (UNICAMP)
Celina M. H. Figueiredo (UFRJ)
Celso C. Ribeiro (PUC/RJ)
Cid C. de Souza (UNICAMP)
Jayme L. Szwarcfiter (UFRJ)
João Meidanis (UNICAMP)
João Carlos Setubal (UNICAMP)
José Augusto R. Soares (USP)
José Coelho de Pina Jr. (USP)
Maria Angela Gurgel (USP)
Márcia Cerioli (UFRJ)
Marcus V. Poggi de Aragão (UNICAMP)
Oscar Porto (PUC/RJ)
Paulo Feofiloff (USP)
Ricardo Dahab (UNICAMP)
Sulamita Klein (UFRJ)
Yoshiharu Kohayakawa (USP)
Yoshiko Wakabayashi (USP) - coordinator

Papers:*

*Note: due to changes in font size and format, the page numbers in this printing of the report are slightly different from those in the original hardcopy issue.

A Pretty Class of Perfect Graphs

Frédéric Maffray
CNRS, LSD2-IMAG
Bôite Postale 53
38041 Grenoble Cedex 9, France
frederic.maffray@imag.fr

Oscar Porto
Dept. de Eng. Elétrica, PUC-Rio
Rua Marquês de São Vicente 225
Predio Cardeal Leme, Sala 401
22453 Rio de Janeiro, RJ, Brasil
oscar@ele.puc-rio.br

Myriam Preissmann
CNRS, ARTEMIS-IMAG
Bôite Postale 53
38041 Grenoble Cedex 9, France
myriam.preissmann@imag.fr

Abstract: We consider the class of graphs where every induced subgraph possesses a vertex whose neighbourhood has no P_4 and no $2K_2$. We prove that Berge's Strong Perfect Graph Conjecture holds for such graphs. The class generalizes several well-known families of perfect graphs, such as triangulated graphs and bipartite graphs. We give a polynomial-time algorithm for optimally coloring the vertices of such a graph.

Introduction

A graph is *perfect* if the vertices of any induced subgraph H can be colored, in such a way that no two adjacent vertices receive the same color, with a number of colors not exceeding the cardinality $\omega(H)$ of a maximum clique of H. Berge introduced perfect graphs in the early 1960s (see [1, 2, 3]) and conjectured that a graph is perfect if and only if it does not contain an induced subgraph isomorphic to a chordless cycle with at least five vertices (an "odd hole") or to the complement of such a cycle (an "odd antihole"). This problem, known as the Strong Perfect Graph Conjecture, is still open and only special cases have been solved (see e.g., [4, 12]). In this paper we prove this conjecture for the graphs where every induced subgraph has a vertex whose neighbourhood has no induced $2K_2$ or P_4. We show that this class contains several well-known classes of perfect graphs such as i-triangulated graphs. We also present an algorithm that finds an optimal coloring and a maximum clique in such graphs. Our proofs exploit some properties of minimally imperfect graphs. Our algorithm is essentially sequential with bichromatic exchanges.

Recall that a vertex is called *simplicial* if its neighbourhood induces a clique. We will say that a vertex is *pretty* if its neighbourhood contains no $2K_2$ (the graph with four vertices and exactly two, non-incident, edges) and no P_4 (the chordless path on four vertices). Clearly, every simplicial vertex is pretty. We then say that a graph is *pretty* if every induced subgraph has a pretty vertex. Our main result is that pretty graphs satisfy the Strong Perfect Graph Conjecture, in other words (since no odd antihole contain a pretty vertex):

Theorem 1 *Every pretty graph with no odd hole is perfect.*

Moreover,

Theorem 2 *Every pretty graph with no odd hole can be optimally colored in polynomial time.*

The proof of this theorem will consist in a polynomial-time coloring algorithm for all pretty perfect graphs.

A graph is called *triangulated* if it has no induced cycle of length at least four. Hajnal and Surányi [10] and Berge [2] proved that triangulated graphs are perfect. We call a graph incomplete if it is not a clique. Dirac [7] proved that every incomplete triangulated graph has two non-adjacent simplicial vertices.

A graph is called *i-triangulated* if every odd cycle of length at least five has two non-crossing chords. It is easy to see that every triangulated graph and every bipartite graph is i-triangulated.

A clique-cutset in a graph G is a subset of vertices S such that $G - S$ is not connected and S is a clique. Given k graphs $G_1, \ldots, G_k$ with disjoint vertex-sets, the *join* of these graphs is the graph obtained by adding all edges with endpoints in different G_i's. A graph is called *clique-separable* if every induced subgraph either has a clique-cutset or is of one among two basic types: (1) the join of a bipartite graph and a clique; (2) the join of several edgeless graphs. Gallai [8] proved that every i-triangulated graph is clique-separable. We will prove:

Theorem 3 *Every clique-separable graph is pretty.*

Before presenting the proof of these results, let us recall some relevant notions about minimally imperfect graphs, and then some useful facts about the structure of graphs with no P_4 and no $2K_2$.

Recall that a graph is called *minimally imperfect* if it is not perfect but all its proper induced subgraphs are perfect. Lovász [11] proved that the complement of any perfect graph is perfect or, equivalently, that the complement of any minimally imperfect graph is minimally imperfect.

Let us say that an edge is *flat* if it lies in no triangle. Tucker [15] proved the following result.

Lemma 1 ([15]) *Assume G is a graph containing no odd hole, xy is a flat edge and is not the only edge of G. Then G is q-colorable if and only if $G - xy$ is q-colorable. Moreover a q-coloring of G can be obtained from any q-coloring of $G - xy$ in polynomial time by a bichromatic exchange.*

Proof. . Clearly if G is q-colorable then $G - xy$ is q colorable. For the converse, consider any q-coloring of $G - xy$. We may assume that x, y have the same color 1, or else we are done. Then we assign to x an arbitrary color 2 used in $G - xy$ as $\omega(G - xy) > 1$, and perform the color exchange on the $(1, 2)$-bichromatic component containing x. This exchange will not reach y, for otherwise there would exist an odd chordless bichromatic path between x and

y, which together with edge xy would form an odd hole in G. Now x and y have different colors, and we have a q-coloring of G. □

An easy corollary of Lemma 1 is:

Lemma 2 ([15]) *If H is a minimally imperfect graph different from an odd hole, and e is a flat edge, then $H - e$ is minimally imperfect.* □

A *star-cutset* of a graph is a disconnecting set C (i.e., $G - C$ has at least two connected components) containing a vertex x adjacent to all vertices of $C - x$. Chvátal ([5]) proved:

Lemma 3 ([5]) *No minimal imperfect graph has a star-cutset.* □

Trivially Perfect Graphs

In order to study pretty graphs, it is useful to determine the structure of graphs with no P_4 and $2K_2$. Equivalently one may look at their complements, i.e., graphs with no induced P_4 and C_4. Such graphs are called *trivially perfect* (see [9, ex. 17, p. 233]). Call a graph *trivial* if it has only one vertex. Call a vertex u of a graph G *universal* if it is adjacent to all vertices of $G - u$. Call a vertex *isolated* if it has no neighbour. It is easily proved that in a connected trivially perfect graph any vertex of maximum degree is universal. Hence every connected trivially perfect graph has a universal vertex. This can be reformulated as follows:

Lemma 4 *Let M be any subgraph of a P_4-free and $2K_2$-free graph. Then either M is the join of several graphs or M has an isolated vertex.* □

From the algorithmic point of view, we will use the algorithm in [6]. This algorithm decides in linear time $O(m)$ if any input graph with m edges is P_4-free and, if yes, produces a tree representation of the graph, which we describe recursively as follows.

- The leaves of the tree are the vertices of the graph and are unlabelled;
- If the graph is disconnected, with components $G_1, \ldots, G_k$, the tree is obtained by creating a 0-labelled root and by linking it to the roots of the trees that represent the graphs $G_1, \ldots, G_k$;
- If the complement $\overline{G}$ of the graph is disconnected, with components $\overline{G}_1, \ldots, \overline{G}_k$, the tree is obtained by creating a 1-labelled root and by linking it to the roots of the trees that represent the graphs $G_1, \ldots, G_k$.

By a theorem of Seinsche [14], a graph is P_4-free if and only if every induced subgraph with at least two vertices either is disconnected or its complement is disconnected. Hence the above tree is well defined and unique. Its size is $O(n)$. Moreover, one can easily compute $\omega(G)$ using the tree representation. Indeed, if the root is labelled 0, the graph is not connected and so $\omega(G) = \max\{\omega(G_1), \ldots, \omega(G_k)\}$, where $G_1, \ldots, G_k$ are the connected components of G. If the root if labelled 1 then $\overline{G}$ is not connected, and so $\omega(G) = \omega(G_1) + \cdots + \omega(G_k)$, where $\overline{G}_1, \ldots, \overline{G}_k$ are the connected components of $\overline{G}$. So $\omega(G)$ can be computed recursively, given the tree, in time $O(n)$.

As for testing $2K_2$-freeness, the following lemma will be useful.

Lemma 5 *Let G be a P_4-free graph and T the associated tree. Then G is $2K_2$-free if and only if, for every 0-node t in T, all children of t except maybe one are leaves of the tree.*

This is a mere reformulation of Lemma 4. □

So we can first test if a graph is P_4-free in linear time and, if it is, test again in linear time if it is $2K_2$-free.

It then follows that we can decide easily if a given graph G is pretty, as follows. For each vertex, test if its neighbourhood is P_4-free and $2K_2$-free. If some vertex v_1 is pretty, iterate the procedure with $G - v_1$. If at step $(i+1)$ no pretty vertex can be found then the induced graph $G - \{v_1, \ldots, v_i\}$ certifies that G is not pretty. If on the contrary the procedure goes through the whole graph, then we obtain a labelling $v_1, v_2, \ldots, v_n$ of the vertex-set of G, where v_i is a pretty vertex of $G - \{v_1, \ldots, v_{i-1}\}$ for each $i = 1, \ldots, n$. We will call this a *pretty ordering*. This procedure finds either a pretty elimination ordering for G or a subgraph that admits no pretty vertex, in time $O(n^2 m)$ if G has n vertices and m edges. Conversely, the existence of a pretty ordering is a certificate that G is pretty. Indeed, if H is any induced subgraph of G, and i is the smallest integer such that v_i is a vertex of H, then it is easy to see that v_i is a pretty vertex of H.

It also follows that we can easily compute the maximum clique size of such a graph. Indeed, if v is any vertex then obviously $\omega(G) = \max\{\omega(G - v), 1 + \omega(N(v))\}$. When v is a pretty vertex then $\omega(N(v))$ can be computed using the tree representing $N(v)$. Hence, iterating this procedure along the pretty elimination ordering, we can find the value of $\omega(G)$ as well as a clique of that size.

The Proof of Theorem 1

First we will prove:

Lemma 6 *No minimally imperfect graph different from an odd hole has a pretty vertex.*

Proof. Let H be a minimally imperfect graph with no odd hole, and assume that H has a pretty vertex x. If $\omega(H) = 2$ then clearly H must be an odd hole, so we may assume $\omega(H) \geq 3$. Call M the subgraph of H induced by the neighbourhood of x. By Lemma 4, either M is a join of several graphs or it has an isolated vertex.
Case 1. M is the join of several graphs. This means that in the minimally imperfect graph $\overline{H}$ vertex x together with its neighbours in $\overline{H}$ form a star-cutset S centered at x, the components of $\overline{H} - S$ being the components of $\overline{M}$. This contradicts Lemma 3.
Case 2. M has an isolated vertex. Let Y be the set of all isolated vertices of M. For each $y \in Y$, the edge xy is a flat edge of H. Consider the graph $H' = H - \{xy \mid y \in Y\}$. By Lemma 2, the graph H' is minimally imperfect. Notice that x is a pretty vertex of H'. Then H' satisfies the hypothesis of Case 1, leading again to a contradiction. □

Now Theorem 1 follows immediately from Lemma 6. □

Coloring Pretty Perfect Graphs

In this section we show how to obtain an ω-coloring of the vertices of any pretty perfect graph G.

Lemma 7 *Let G be a graph that contains no odd hole and admits a pretty vertex v, of degree $d(v)$. Then from any $\omega(G-v)$-coloring of $G-v$ we can deduce, through a sequence of at most $d(v)$ bichromatic exchanges, an $\omega(G)$-coloring of G.*

Proof. Let f be any $\omega(G)$-coloring of $G-v$. The following algorithm will produce an $\omega(G)$-coloring of G.

Begin Determine the number $|f(N(v))|$ of colors used by f in the neighbourhood of v, and compute $\omega(N(v))$.

- If $\omega(N(v)) = \omega(G-v)$, then give a new color to v, and stop.
- If $|f(N(v))| < \omega(G-v)$ then assign to v any color in $f(V-v) - f(N(v))$, and stop.
- Set $i := 1$ and $N_1 := N(v)$. Let I_1 be the set of isolated vertices in N_1, and $J_1 := N_1 - I_1$.

Step 1. If $f(I_i) \subseteq f(J_i)$ then let J' be a component of $\overline{J_i}$ such that $\omega(J') < |f(J')|$. Set $N_{i+1} := J'$; let I_{i+1} be the set of isolated vertices in N_{i+1}, and $J_{i+1} := N_{i+1} - I_{i+1}$. Set $i := i+1$ and iterate Step 1.

Step 2. Here $f(I_i) - f(J_i) \neq \emptyset$. Let c be a color which appears in $f(I_i) - f(J_i)$, and let c' be any color in $f(N_i) - \{c\}$. Assign the color c to v. Perform the bichromatic exchange on colors c and c' starting from every neighbour of v of color c.

End

Let us prove that this algorithm properly colors G with $\omega(G)$ colors. This is clear if the algorithm stops before Step 1. Now assume the algorithm goes to Step 1, and consider the beginning of any execution of this step. Note that $f(I_i) \subseteq f(J_i)$ implies that $f(J_i) = f(N_i)$. However, recall that $|f(N_i)| > \omega(N_i)$; indeed, this is obviously true when $i = 1$, and it is true when $i > 1$ by the choice of $N_i = J'$ made during the previous execution of Step 1. Also, $\omega(N_i) = \omega(J_i)$ by the definition of I_i and J_i. Hence the set J' exists.
Now consider the beginning of the first execution of Step 2. The color c exists trivially, and the color c' exists because obviously $|f(N_i)| \geq 2$. There remains to prove that the bichromatic exchanges performed in Step 2 do produce a proper coloring of G. Since G has no odd hole, it suffices to show that v does not have two adjacent neighbours of color c and c'. To prove this let us consider any vertex x such that $f(x) \in f(N_i)$. Then necessarily x is either in N_i or in $I_{i-1} \cup \cdots \cup I_2 \cup I_1$. In consequence the neighbours of v of color c are all in $I_i \cup \cdots \cup I_2 \cup I_1$, and the neighbours of v of color c' are all in $N_i \cup I_{i-1} \cup \cdots \cup I_2 \cup I_1$. This implies that there is no edge between two neighbours of v of color c and c', because there is no edge between I_i and N_j whenever $j \geq i$. □

Note that in the proof of this lemma, the numbers that need to be computed are available from the examination of f and of the tree representing $N(v)$. Thus the complexity of this algorithm is $O(m(v) + d(v)n)$, where v is the pretty vertex under consideration and $m(v)$ is the number of edges in the neighbourhood of v.

Corollary 1 *Let G be a graph which has a pretty vertex v and has no odd hole. Then G is perfect if and only if $G-v$ is perfect.* □

Lemma 7 yields an algorithm for ω-coloring every pretty perfect graph, given any pretty elimination ordering. The main subroutine is described in the preceding lemma. Hence the overall complexity is $O(mn)$. This proves Theorem 2.

The Proof of Theorem 3

We will prove a lemma similar to Dirac's [7].

Lemma 8 *In an incomplete clique-separable graph, there exists a pair of non-adjacent pretty vertices.*

Proof. Let G be a clique-separable graph. We prove the lemma by induction on the number of vertices of G.

First assume that G is the join of a bipartite graph B and a clique. Observe that the neighbourhood of any vertex from B is the join of a clique and a stable set, which is easily seen to be $2K_2$-free and P_4-free. Since G is incomplete, there are two non-adjacent vertices in B. Then these two vertices form the desired pair.

Next, assume that G is the join of several edgeless graphs $G_1, \ldots, G_k$. The neighbourhood of any vertex is the join of $k-1$ of these edgeless graphs, which again is easily seen to be $2K_2$-free and P_4-free. Since G is incomplete we may assume that G_1 has at least two vertices. Then these two vertices form the desired pair.

Finally, assume that G has a clique cutset C. Let $R_1, \ldots, R_k$ be the components of $G - C$, with $k \geq 2$. For $i = 1, 2$, the subgraph induced by $R_i \cup C$ either is a clique or has, by induction, two non-adjacent pretty vertices of which at least one, say x_i, is necessarily in R_i. If $R_i \cup C$ is a clique pick any x_i in R_i. Then x_1, x_2 is the desired pair for G. □

Now Theorem 3 follows immediately from Lemma 8. □

As a conclusion, we may want to compare pretty perfect graphs with other known classes. It is easy to check that the line-graph of $K_{3,3} - e$ is a pretty perfect graph; however it is neither in the class BIP* defined in [5] nor in the class of quasi-parity graphs defined in [13] which contain most classical families of perfect graphs.

We leave as an open problem the recognition of pretty perfect graphs.

References

[1] C. BERGE. Les problèmes de coloration en théorie des graphes. *Publ. Inst. Stat. Univ. Paris*, 9:123–160, 1960.

[2] C. BERGE. Färbung von Graphen, deren sämtliche bzw. deren ungerade Kreise starr sind (Zusammenfassung). *Wiss. Z. Martin Luther Univ. Math.-Natur. Reihe*, 10:114–115, 1961.

[3] C. BERGE. *Graphs.* North-Holland, Amsterdam/New York, 1985.

[4] C. BERGE and V. CHVÁTAL, editors. *Topics on perfect graphs (Annals of Disc. Math. 21).* North-Holland, Amsterdam, 1984.

[5] V. CHVÁTAL. Star-cutsets and perfect graphs. *J. Comb. Theory B*, 39:189–199, 1985.

[6] D.G. CORNEIL, Y. PERL, and L.K. STEWART. A linear recognition algorithm for cographs. *SIAM J. Comput.*, 14:926–934, 1985.

[7] G.A. DIRAC. On rigid circuit graphs. *Abh. Math. Sem. Univ. Hamburg*, 25:71–76, 1961.

[8] T. GALLAI. Graphen mit triangulierbaren ungeraden Vielecken. *Magyar Tud. Akad. Mat. Kutato Int. Közl.*, 7:3–36, 1962.

[9] M.C. GOLUMBIC. Trivially perfect graphs. *Disc. Math.*, 24:105–107, 1978.

[10] A. HAJNAL and J. SURÁNYI. Über die Auflösung von Graphen in vollständige Teilgraphen. *Ann. Univ. Sc. Budapestinensis*, 1:113–121, 1958.

[11] L. LOVÁSZ. A characterization of perfect graphs. *J. Comb. Theory B*, 13:95–98, 1972.

[12] L. LOVÁSZ. Perfect graphs. In L.W. Beineke and R.J. Wilson, editors, *Selected Topics in Graph Theory 2*, pages 55–87. Academic Press, 1983.

[13] H. MEYNIEL. A new property of critical imperfect graphs and some consequences. *European Journal of Combinatorics*, 8:313–316, 1987.

[14] S. SEINSCHE. On a property of the class of n-colorable graphs. *J. Comb. Th. B*, 16:191–193, 1974.

[15] A. TUCKER. Coloring perfect $(K_4 - e)$-free graphs. *J. Comb. Th. B*, 43:313–318, 1987.

A Scalable Parallel Algorithm for List Ranking

Frank Dehne*
School of Computer Science
Carleton University
Ottawa, Canada K1S 5B6
dehne@scs.carleton.ca

Siang W. Song*
Dept. de Ciência da Computação
IME- Universidade de São Paulo
05508-900 São Paulo, SP, Brasil
song@ime.usp.br

Abstract: We present a scalable parallel algorithm for distributed memory multiprocessors to solve the problem of list ranking. For an n-element linked list we assume that each of the p processors has $O(n/p)$ local memory. The algorithm consists of a number of alternating computing and communication rounds. In each communication round, each processor can send $O(n/p)$ data and receive $O(n/p)$ data. By using a conveniently chosen sample of $O(n/p)$ pivot elements we present an efficient algorithm. It requires only $r \leq (4k+6)\log(\frac{2}{3}p)+8 = \tilde{O}(k \log p)$ communication rounds, with high probability, where $k < \ln^*(n)$ is an extremely small number.

1 Introduction

The problem of list ranking consists of finding the location of each element in a linked list with respect to the end of the list. It appears as subproblem in several graph and tree problems. Therefore an efficient solution for list ranking has direct consequences in other applications.

Optimal $O(\log n)$ EREW PRAM algorithms are proposed in [5]. Parallel list ranking algorithms using randomization were proposed in [1, 6].

We present a scalable parallel algorithm for the n-element list ranking problem using p processors. The algorithm requires only $r \leq (4k+6)\log(\frac{2}{3}p)+8 = \tilde{O}(k \log p)$ communication rounds, with high probability, where $k < \ln^*(n)$ is an extremely small number.

2 Random Sampling in Linear Linked Lists

Consider a linear linked list with a set S of n nodes. In this section we will show that if we select $\frac{n}{p}$ random elements (pivots) of S then, with high probability, these pivots will split S into sublists whose maximum size is bound by $3p\ln(n)$.

*Research partially supported by the Natural Sciences and Engineering Research Council of Canada, FAPESP (Fundação de Amparo à Pesquisa do Estado de São Paulo, Proc. No. 93/0603-1, 94/4544-2, 95/0767-0, 95/1367-5), CNPq (Conselho Nacional de Desenvolvimento Científico e Tecnológico, Proc. No. 523112/94-7 and PROTEM-2-TCPAC Proc. No. 680060/94-4), and the Commission of the European Communities (Project ITDC-207).

Theorem 1 *$\frac{n}{p}$ randomly chosen pivots partition S into $\frac{n}{p}+1$ sublists S_j with $m = \max_{0\leq j\leq p} |S_j|$ such that*

$$Prob\{m \geq c3p\ln(n)\} \leq \frac{1}{n^c}, c > 2$$

3 A Simple Algorithm Using A Single Random Sample

Consider a random set $S' \subset S$ of pivots. For each $x \in S$ let $nextPivot(x, S')$ refer to the closest pivot following x in the list S. (W.l.o.g. assume that the last element, λ, of S is selected as a pivot and let $nextPivot(\lambda, S') = \lambda$. Note that for $x \neq \lambda$, $nextPivot(x, S') \neq x$.) Let $distToPivot(x, S')$ be the distance between x and $nextPivot(x, S')$ in list S. Furthermore, let $m(S, S') = \max_{x\in S} distToPivot(x, S')$.

The *modified pointer jumping problem* for S with respect to S' refers to the problem of determining for each $x \in S$ its next pivot $nextPivot(x, S')$ as well as the distance $distToPivot(x, S')$. The input/output structure for the modified pointer jumping problem is the same as for the pointer jumping problem.

Algorithm 1

(1) Select a set $S' \subset S$ of $\tilde{O}(\frac{n}{p})$ random pivots as follows: Every processor P_i makes for each $x \in S$ stored at P_i an independent biased coin flip which selects x as a pivot with probability $\frac{1}{p}$.

(2) All processors solve collectively the *modified pointer jumping problem* for S with respect to S' (details will be discussed later).

(3) Using an all-to-all broadcast, the values $nextPivot(x, S')$ and $distToPivot(x, S')$ for all pivots $x \in S'$ are broadcast to all processors.

(4) Using the data received in Step 3, each processor P_i can solve the pointer jumping problem for the nodes stored at P_i sequentially in time $\tilde{O}(\frac{n}{p})$.

— End of Algorithm —

Theorem 2 *Algorithm 1 solves the pointer jumping problem using, with high probability, at most $1 + \log(3p) + \log\ln(n)$ communication rounds and $\tilde{O}(\frac{n}{p})$ sequential computing time.*

Corollary 2 *If $\frac{n}{p} \leq e^{(3p)^\alpha}$, for some constant $\alpha > 1$, then the number of communication rounds required by Algorithm 1 is bounded by $2 + (\alpha + 1)\log(3p) = \tilde{O}(\log p)$.*

4 Improving The Maximum Sublist Size

We will now present an algorithm which improves the maximum sublist size obtained in Algorithm 1 and solves the pointer jumping problem by using, with high probability, $r \leq (4k + 6)\log(\frac{2}{3}p) + 8$ communication rounds and $\tilde{O}(\frac{n}{p})$ sequential computing time where

$$k := \min\{i \geq 0 | \ln^{(i+1)} n \leq (\frac{2}{3}p)^{2i+1}\}.$$

Note that $k < \ln^*(n)$ is an extremely small number.

Algorithm 2

(1) Perform Step 1 of Algorithm 1. Mark all selected pivots *black* and all other nodes *white*.

(2) For $i = 1, \ldots, k$ do

(2a) For each *black* node x, all nodes which are to the right of x (in list S) and have distance at most $\frac{2}{3}p$ are marked *red*. Note: previously *black* nodes (pivots) that are now marked *red* are no longer considered pivots.

(2b) For each *black* node x, all nodes which are to the left of x (in list S) and have distance at most $\frac{2}{3}p$ are marked *red*.

(2c) Every processor P_i makes for each *white* node $x \in S$ stored at P_i an independent biased coin flip which selects x as a new pivot, and marks it *black*, with probability $\frac{1}{p}$.

(2d) Every processor P_i marks *white* every *red* node $x \in S$ stored at P_i.

(3) Let $S' \in S$ be the subset of *black* nodes obtained after Step 2. Continue with Steps 2 – 4 of Algorithm 1.

— End of Algorithm —

Theorem 3 *With high probability, Algorithm 2 solves the pointer jumping problem with* $r \leq (4k+6)\log(\frac{2}{3}p)+8 = \tilde{O}(k \log p)$ *communication rounds and* $\tilde{O}(\frac{n}{p})$ *sequential computing time.*

References

[1] J. R. Anderson and G. L. Miller, "A simple randomized parallel algorithm for list ranking". *Information Processing Letters*, Vol. 33, No. 5, January 1990, pp. 269 – 273.

[2] M. J. Atallah, S. E. Hambrusch, "Solving tree problems on a mesh-connected processor array," *Information adn Control*, Vol. 69, 1986, pp. 168-187.

[3] S. Baase, "Introduction to parallel connectivity, list ranking, and Euler tour techniques". J. H. Reif (ed.) *Synthesis of Parallel Algorithms*. Morgan Kaufmann Publisher, 1993.

[4] Belloch *et. al.*, "A comparison of sorting algorithms for the Connection Machine CM-5," in Proc. SPAA'91 - *3rd Annual ACM Symposium on Parallel Algorithms and Architectures*, 1991, pp. 3 – 16.

[5] R. Cole and U. Vishkin, "Approximate parallel scheduling, Part I: the basic technique with applications to optimal parallel list ranking in logarithmic time. *SIAM J. Computing*, Vol. 17, No. 1, 1988, pp. 128 – 142.

[6] G. L. Miller and J. H. Reif, "Parallel tree contraction part 1: Further applications". *SIAM J. Computing*, Vol. 20, No. 6, December 1991, pp. 1128 – 1147.

The Homogeneous Set Sandwich Problem

Celina M. H. de Figueiredo*
Sulamita Klein*
Instituto de Matemática, UFRJ
Caixa Postal 68530
21944 Rio de Janeiro, RJ, Brasil
{celina,sula}@cos.ufrj.br

Hazel Everett*
Dep. d'Informatique
Université du Québec à Montréal
Montréal, Québec, H3C 3P8, Canada
everett@math.uqam.ca

Márcia R. Cerioli*
COPPE - Progr. de Eng. de Sistemas e Computação
Universidade Federal do Rio de Janeiro
Caixa Postal 68511
21945-970 Rio de Janeiro, RJ, Brasil
cerioli@cos.ufrj.br

Abstract: The graph sandwich problem for property Φ is defined as follows: Given two graphs $G_1 = (V, E_1)$ and $G_2 = (V, E_2)$ such that $E_1 \subseteq E_2$, is there a graph $G = (V, E)$ such that $E_1 \subseteq E \subseteq E_2$ which satisfies property Φ? We present a polynomial-time algorithm for solving the graph sandwich problem, when property Φ is to contain a homogeneous set.

1 Introduction

We say that a graph $G_1 = (V, E_1)$ is a *spanning* subgraph of $G_2 = (V, E_2)$, if $E_1 \subseteq E_2$ and we say that a graph $G = (V, E)$ is a *sandwich* graph for the pair G_1, G_2, if $E_1 \subseteq E \subseteq E_2$. According to [5], the GRAPH SANDWICH PROBLEM FOR PROPERTY Φ is defined as follows:

Problem: GRAPH SANDWICH PROBLEM FOR PROPERTY Φ
Instance: Two graphs, G_1 and G_2, such that G_1 is a spanning subgraph of G_2.
Question: Does there exist a sandwich graph for the pair G_1, G_2 which satisfies property Φ?

Note that graph sandwich problems arise as a natural generalization of recognition problems. The recognition problem for a class of graphs $\mathcal{C}$ is equivalent to the graph sandwich problem in which $G_1 = G_2 = G$, where G is the graph we want to recognize and property Φ is to belong to class $\mathcal{C}$.

In [5], several subclasses of perfect graphs are considered with respect to sandwich problems. In particular, they prove that the GRAPH SANDWICH PROBLEM FOR SPLIT GRAPHS remains in P. On the other hand, they prove that the GRAPH SANDWICH PROBLEM FOR PERMUTATION GRAPHS turns out to be NP-complete.

*This research was partially supported by NSERC and CNPq/ProtemCC I, ProtemCC II.

We are interested in graph sandwich problems for properties Φ which are related to decompositions that arise in perfect graph theory. Two such properties are: to contain a homogeneous set and to contain a clique cutset. In this paper we present a polynomial-time algorithm for solving the graph sandwich problem, when property Φ is to contain a homogeneous set.

2 Homogeneous set

By a *homogeneous set* in a graph $G = (V, E)$, we shall mean a set H of vertices of G such that each vertex of $V \setminus H$ is either adjacent to all vertices of H or to none of the vertices of H. In order not to have a trivial definition, we also ask that $|H| \geq 2$ and $|V \setminus H| \geq 1$.

We give in Figure 1 below two examples: a graph with a homogeneous set and a graph with no homogeneous set. We note that, by definition, any complete bipartite graph with at least three vertices admits a homogeneous set. On the other hand, any chordless cycle with at least five vertices does not admit a homogeneous set.

Figure 1: Complete bipartite graph and chordless cycle.

We observe that the existence of a homogeneous set H in a graph G implies that the set of vertices $V \setminus H$ can be partitioned into two sets A and N such that:

$$\begin{aligned} A &= \{v \in V : \mathcal{N}(v) \cap H = H\}; \\ N &= \{v \in V : \mathcal{N}(v) \cap H = \emptyset\}, \end{aligned}$$

where $\mathcal{N}(v)$ denotes the neighbourhood of v, i.e., the set of vertices of V that are adjacent to v. We also say that a vertex which is in the neighbourhood of v *sees* v. We extend this terminology to sets, by saying that vertex v sees the set $\mathcal{N}(v)$. In addition, vertex v *sees partially* set S, if v sees at least a vertex of S but not all vertices of S. We shall denote by S both the vertex set $S \subset V$ and $G[S]$, the subgraph of G induced by S.

We can represent a graph G with a homogeneous set H by the diagram in Figure 2, where a continuous line between two sets represents the property that each vertex of one set is adjacent to each vertex of the other set. A broken line represents the property that no vertex of a set is adjacent to any vertex of the other set.

The property of containing a homogeneous set is a self-complementary property. A central result in perfect graph theory shows that the class of perfect graphs is a self-complementary class, i.e., a graph is perfect if and only if its complement is perfect. In

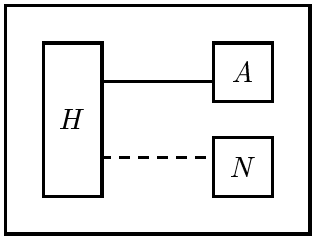

Figure 2: Graph G with homogeneous set H.

the proof of the perfect graph theorem, Lovász used the concept of homogeneous set to define a decomposition procedure that preserves perfection [7].

Given two disjoint graphs G and H, and given a vertex x of G consider the following graph G': Take the disjoint union of $G \setminus x$ and H, and for each pair of vertices y, z with $y \in G \setminus x$ and $z \in H$, add the edge $(y, z) \in E(G')$ if and only if $(x, y) \in E(G)$. In this case, we say that G' was obtained by *substitution* of x by H in G. Figure 3 represents a graph obtained by substitution.

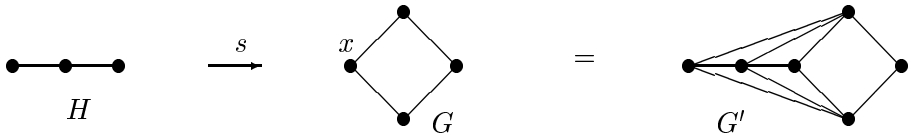

Figure 3: Substitution.

We note that a graph admits a homogeneous set if and only if this graph can be obtained by substitution. We also note that by using substitution, we have a natural decomposition of a given root graph G with homogeneous set H into two subgraphs: H itself and $G \setminus (H \setminus x)$. This decomposition can be used to compute both the chromatic number and the clique number of G, given those parameters for its children [2]. Moreover, this decomposition has been used to obtain recognition algorithms for well known classes of perfect graphs such as comparability graphs [4] and bull-free perfect graphs [10].

Polynomial-time algorithms for finding homogeneous sets are given in [9], [8], [11], [1] and [6]. The fastest is an unpublished $O(m)$-algorithm by Spinrad. See [11] for an $O(m\alpha(m, n))$-algorithm.

We note that the sandwich problem for homogeneous set is not trivial, in the sense that it is neither a hereditary property (in this case the graph G_1 is always a solution) nor an ancestral property (in this case the graph G_2 is always a solution). Figure 4 shows an instance where neither G_1 nor G_2 admit a homogeneous set but there exists a sandwich graph G for this property with $H = \{3, 4, 5\}$, $A = \{2, 6\}$ and $N = \{1\}$.

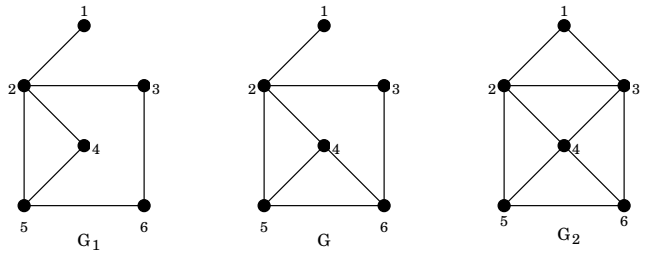

Figure 4: Sandwich graph G that admits a homogeneous set.

3 The Algorithm

Let $G_1 = (V, E_1)$ and $G_2 = (V, E_2)$ be two graphs such that G_1 is a spanning subgraph of G_2. Our algorithm tests whether there exists a sandwich graph G that admits a homogeneous set. It follows closely the algorithm for testing the existence of a homogeneous set in a graph G given by Reed [9].

The idea is to test, for any two vertices x and y in V, if there exists a sandwich graph G that admits a homogeneous set H with x and y in H.

Our method starts with a candidate set H consisting of the vertices x and y. At each step, we add new vertices to the current set H by considering adjacency with respect to both G_1 and G_2. For the current set H, we partition $V \setminus H$ into four sets: A, N, Q and C. This partition is based in two properties: a vertex that is adjacent to at least a vertex of H in G_1 is also adjacent to at least a vertex of H in G_2 and therefore cannot be placed in set N. Analogously, a vertex that is not adjacent to at least one vertex of H in G_2 is not adjacent to at least one vertex of H in G_1 either and therefore cannot be placed in set A.

Thus, we consider first a vertex that is adjacent to at least a vertex of H in G_1 and it is adjacent to all vertices of H in G_2. We place this vertex in set A. This means that this vertex cannot be in set N, i.e., in case there is a solution, this vertex is in $H \cup A$. A vertex that is adjacent to no vertex of H in G_1 and it is not adjacent to at least one vertex of H in G_2 is placed in set N. This means that this vertex cannot be in set A, i.e., in case there is a solution, this vertex is in $H \cup N$. On the other hand, a vertex that is adjacent to no vertex of H in G_1 but it is adjacent to all vertices of H in G_2 is placed in set Q. This set contains the question mark vertices as we have no information about those vertices yet. We put all the remaining vertices into the last set C. More precisely, a vertex that sees H partially both in G_1 and in G_2, cannot be in A because it is not adjacent to at least one vertex of H in G_2 and cannot be in N because it is adjacent to at least a vertex of H in G_1. Thus, such a vertex is placed in C. The next set H is obtained by adding C to the current set H.

We have below the complete algorithm and the corresponding proof of correctness. We shall denote by $\mathcal{N}_i(v)$ the neighbourhood of v respectively in G_i, $i \in \{1, 2\}$.

Algorithm 3 *Test for the existence of a sandwich graph G that admits a homogeneous set H containing two given vertices.*

Input: $G_1 = (V, E_1)$, $G_2 = (V, E_2)$ such that $E_1 \subseteq E_2$ and two vertices $x, y \in V$.

Output: YES or NO.

If the answer is YES, the algorithm also gives the edge set E of the sandwich graph G and the partition of the vertex set V into H, A, and N such that

$$\begin{aligned} H &= \text{homogeneous set of } G \text{ containing } x, y; \\ A &= \{v \in V : \mathcal{N}(v) \cap H = H\}; \\ N &= \{v \in V : \mathcal{N}(v) \cap H = \emptyset\}. \end{aligned}$$

1. Create five vertex sets: H, A, N, Q and C.
2. $H := \{x, y\}$;
   Partition the vertex set $V^\star = V \setminus \{x, y\}$ into the following four sets:
   $A := [\mathcal{N}_1(x) \cup \mathcal{N}_1(y)] \cap [\mathcal{N}_2(x) \cap \mathcal{N}_2(y)]$;
   $N := [V^\star \setminus (\mathcal{N}_1(x) \cup \mathcal{N}_1(y))] \cap [(V^\star \setminus \mathcal{N}_2(x)) \cup (V^\star \setminus \mathcal{N}_2(y))]$;
   $Q := [V^\star \setminus (\mathcal{N}_1(x) \cup \mathcal{N}_1(y))] \cap [\mathcal{N}_2(x) \cap \mathcal{N}_2(y)]$;
   $C := V^\star \setminus (A \cup N \cup Q)$.
3. If $H = V$ return NO and stop.
4. If $C = \emptyset$
   $N := N \cup Q$;
   $E := E_1 \cup \{(a, h) : a \in A \text{ and } h \in H\}$;
   return (YES, E, H, A, N) and stop.
5. Consider three auxiliary vertex sets: C', A' and N'.
   $C' := \emptyset$; $A' := \emptyset$; $N' := \emptyset$;
   For each $z \in C$
   $C' := C' \cup (\mathcal{N}_1(z) \cap N) \cup (A \setminus \mathcal{N}_2(z))$;
   $A' := A' \cup (\mathcal{N}_1(z) \cap Q)$;
   $N' := N' \cup (Q \setminus \mathcal{N}_2(z))$.
6. $H := H \cup C$;
   $A := (A \setminus C') \cup (A' \setminus (A' \cap N'))$;
   $N := (N \setminus C') \cup (N' \setminus (A' \cap N'))$;
   $Q := Q \setminus (A' \cup N')$;
   $C := C' \cup (A' \cap N')$;
   Go to 3.

Theorem 1 *Algorithm 3 correctly tests for the existence of a sandwich graph G that admits a homogeneous set H containing two given vertices.*

Proof: The algorithm constructs a sequence of sets

$$\{x, y\} = H_0 \subset H_1 \subset \ldots \subset H_t$$

by adding to the current set H at each step the vertices that must be in H, i.e., the vertices that see partially the current set H, both in G_1 and G_2.

Suppose first that the algorithm stops with $C = \emptyset$. We want to show that the sets E, H, A, N are such that H is a homogeneous set of G. For we shall prove that Step 6

keeps as invariant the following relation of set H with respect to sets A, N, Q and C: set A contains the vertices that cannot be in N; set N contains the vertices that cannot be in A; set Q contains the question mark vertices; set C contains the vertices that cannot be in A and that cannot be in N either.

This is done by introducing the three auxiliary vertex sets C', A' and N' in Step 5. Set A' contains those vertices that are no longer in Q as they now see in G_1 a vertex of H. In Step 6, part of set A' is put in set A and part of set A' is put in set C. Analogously, set N' contains those vertices that are no longer in Q as they now fail to see in G_2 a vertex of H. Finally, set C' contains those vertices that are no longer in N as they now see in G_1 a vertex of H and contains those vertices that are no longer in A as they now fail to see in G_2 a vertex of H.

On the other hand, suppose that the algorithm stops with $H_t = V$ but there exists a sandwich graph G with a homogeneous set F containing $\{x, y\}$. Because F contains $\{x, y\}$ and $F \subset H_t = V$, we must have an index $i > 0$ such that $H_{i-1} \subseteq F$ but $H_i \not\subseteq F$. By construction, $H_i \setminus H_{i-1}$ is the set of vertices that see H_{i-1} partially, both in G_1 and G_2. Now, any vertex in $H_i \setminus F$ is in $H_i \setminus H_{i-1}$. Hence there exists a vertex in $V \setminus F$ that sees F partially, both in G_1 and G_2, a contradiction. ■

Algorithm 3 stops after a polynomial number of steps since the loop of Steps 3–6 is executed at most n times and all statements in the algorithm are based on unions and intersections of finite sets.

We run Algorithm 3 once for every two vertices x and y in V, until we find a sandwich graph with homogeneous set containing the pair x, y or we have tested all pairs of vertices. So Algorithm 3 is executed at most n^2 times to solve the sandwich problem for homogeneous set.

4 Conclusion

We have presented a polynomial-time algorithm for the homogeneous set sandwich problem.

One other type of relaxation for recognition problems which has been well-studied is the EDGE-COMPLETION PROBLEM. It asks, given a graph and an integer k, whether one can add to the original graph at most k edges in order to obtain a graph with a certain property. Such problems have been studied for hamiltonian graphs, interval graphs, path graphs [3] and chordal graphs [12].

The corresponding edge-completion problem for homogeneous set asks for the minimum number of edges needed to be added to a given graph G_1 in order to give a graph with a homogeneous set. We note that our algorithm does not solve this problem. Actually, if G_2 is the complete graph, then our algorithm always returns $H = \{x, y\}$. If G_1 contains a homogeneous set, but contains no homogeneous set with just two vertices, then our algorithm does not give a solution with the minimum number of edges.

Our algorithm does satisfy a minimality property with respect to the number of vertices in H, i.e., our method looks for a sandwich graph G with homogeneous set H (that contains a given pair of vertices) as small as possible.

Acknowledgements

We are grateful to João Meidanis for introducing us to graph sandwich problems and to Jayme L. Szwarcfiter for pointing out the homogeneous set sandwich problem.

References

[1] A. Cournier, *Sur quelques Algorithmes de Décomposition de Graphes*, Thése, Université Montpellier II, (1993).

[2] C. M. H. de Figueiredo, *Um Estudo de Problemas Combinatórios em Grafos Perfeitos*, Doctoral Thesis (in portuguese), Progr. de Eng. de Sistemas e Computação, COPPE/UFRJ (1991).

[3] M. R. Garey, D. S. Johnson, Computers and Intractability, a guide to the theory of NP-completeness, Freeman, San Francisco, (1979) 198–199.

[4] M.C. Golumbic, Comparability graphs and a new matroid, *J. Comb. Theory. (B)* **22** (1977) 68–90.

[5] M. C. Golumbic, H. Kaplan and R. Shamir, Graph Sandwich Problems. Technical Report 270/92, The Moise and Frida Eskenesy Institute of Computer Sciences, Telaviv University, (1992).

[6] S. Klein, *Algoritmos e Complexidade de Decomposição em Grafos*, Doctoral Thesis (in portuguese), Progr. de Eng. de Sistemas e Computação, COPPE/UFRJ (1994).

[7] L. Lovász, Normal hypergraphs and the perfect graph conjecture, *Discrete Math.* **2** (1972) 253–267.

[8] J.H. Muller and J. Spinrad, Incremental modular decomposition, *Journal of the Association for Computing Machinery* **1** (1989) 1–19.

[9] B. Reed, *A Semi-Strong Perfect Graph Theorem*, Ph.D. Thesis, School of Computer Science, McGill University, Montreal (1986).

[10] B. Reed, N. Sbihi, Recognizing Bull-Free Perfect Graphs, *Graphs and Combinatorics* **11** (1995) 171–178.

[11] J. Spinrad, P_4-trees and Substitution Decomposition, *Discrete Applied Mathematics* **39** (1992) 263–291.

[12] M. Yannakakis, Computing the minimum fill-in is NP-complete, *SIAM J. Alg. Disc. Math* **2** (1981) 77–79.

Local Conditions for Edge-Coloring

Celina M. H. de Figueiredo*
Instituto de Matemática
Universidade Federal do Rio de Janeiro
Caixa Postal 68530
21944 Rio de Janeiro, RJ, Brasil
celina@cos.ufrj.br

João Meidanis†
Célia Picinin de Mello†
Universidade Estadual de Campinas
Departamento de Ciência da Computação
Caixa Postal 6065
13081-970 Campinas, SP, Brasil
{meidanis,celia}@dcc.unicamp.br

Abstract: In this note we investigate three versions of the overfull property for graphs and their relation to the edge-coloring problem. Each of these properties implies that the graph cannot be edge-colored with Δ colors, where Δ is the maximum degree. The three versions are nonequivalent for general graphs. However, we show that for the classes of indifference graphs, split graphs, and complete multipartite graphs some equivalences hold.

1 Introduction

Edge-coloring is an important problem in graph theory. It concerns coloring the edges of a graph so that incident edges get different colors. The goal is to do that using the minimum number of colors.

A celebrated theorem by Vizing states that this minimum is always Δ or $\Delta + 1$, where Δ is the maximum degree [11, 7]. To decide between these two possibilities is however NP-hard [6, 1]. More precisely, if we denote by C1 the class of graphs G edge-colorable with $\Delta(G)$ colors, and by C2 the complementary class, we have that C1 $\in$ NP while C2 $\in$ co-NP.

This means that C1 graphs always have short certificates. Indeed, to convince someone that a graph is in C1 all we have to do is to exhibit a Δ-coloring. In contrast, to show that a graph is in C2 we must produce an argument that no Δ-coloring exists.

Nevertheless, for some graphs this argument can be very simple. For instance, the graph in Figure 1 is in C2. This graph is small enough as to permit trying all possibilities, but the following argument is simpler. Notice that, in a valid coloring, each color corresponds to a matching and hence can be assigned to at most two edges in this particular case. Since the total number of edges is five, it becomes evident that $\Delta = 2$ colors do not suffice.

Graphs to which a similar argument can be applied have been termed **overfull** [8, 9, 4, 3]. We say that a graph G is **overfull** when the number of vertices n is odd and

$$\Delta \frac{n-1}{2} < m,$$

*Partially supported by CNPq, grant 30 1160/91.0 and ProTeM-CC.
†Partially supported by FAPESP and CNPq.

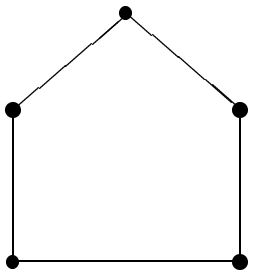

Figure 1: An overfull graph.

where m is the number of edges.

Notice that we are using the fact that a matching in an n-vertex graph has size at most $(n-1)/2$ when n is odd. The matching number of the graph could be used instead of the expression $(n-1)/2$, but we will concentrate in the traditional definition — the one given above — in this note.

In addition, observe that this argument will not work for a graph with even n. In this case, the following relation is always true:

$$\Delta \frac{n}{2} \geq m.$$

This is just another way of saying that the maximum degree is not smaller than the average degree, if we recall that $2m$ is equal to the sum of all degrees.

In our studies on edge-coloring we found that two other definitions of overfullness are also of interest. We state them in the sequel.

A graph G is **subgraph-overfull** when there is an overfull subgraph H of G with $\Delta(H) = \Delta(G)$. Here, a subgraph is formed by taking some of the vertices and some of the edges of G. It does not need to be an induced subgraph. However, considering induced subgraphs leads to an equivalent definition.

Notice that subgraph-overfull graphs are in C2, since we need at least $\Delta(G)+1$ colors just for the edges in H.

If the overfull subgraph H happens to be a **neighborhood**, that is, is induced by a Δ-vertex and all its neighbors, then we call G **neighborhood-overfull**.

Again, neighborhood-overfull graphs are in C2.

In the rest of this note we will denote by O, SO, and NO the classes of overfull, subgraph-overfull, and neighborhood-overfull graphs, respectively.

Theorem 1 *We have*

$$O \subseteq SO \subseteq C2$$

and

$$NO \subseteq SO \subseteq C2.$$

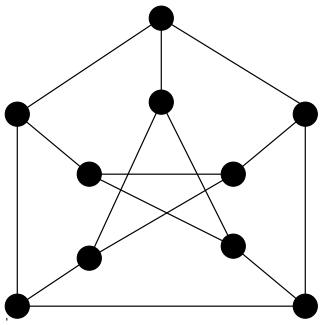

Figure 2: Petersen graph.

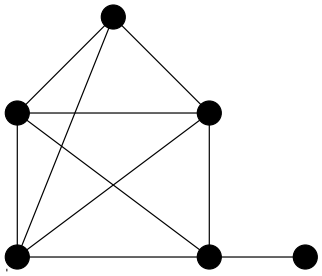

Figure 3: A neighborhood-overfull graph which is not overfull.

The proof is immediate from the definitions.

All inclusions in Theorem 1 are proper. In addition, O and NO are incomparable. The Petersen graph in Figure 2 is an example of a C2 graph which is not subgraph-overfull. Actually the Petersen graph with one vertex removed is a C2 graph which is not subgraph-overfull.

To get an example of a subgraph-overfull graph that is not overfull itself, consider the split graph F depicted in Figure 3. This graph can be partitioned into a clique of size 4 and a stable set of size 2. One vertex of the stable set sees three vertices of the clique and the other vertex of the stable set sees the forth vertex of the clique. This graph is not overfull as it contains an even number of vertices. On the other hand, by removing its vertex of degree one, we obtain a copy of $K_5 \setminus$ one edge, an overfull graph, with the same maximum degree as F. We note that the graph $K_5 \setminus$ one edge is actually neighborhood-overfull as it contains a universal vertex. Thus graph F is an example of a neighborhood-overfull graph that is not overfull.

Finally, Figure 4 shows an example of a graph that is overfull but not neighborhood-overfull. Thus this graph is a subgraph-overfull graph that is not neighborhood-overfull. We note that a graph with an odd maximum degree cannot be neighborhood-overfull.

The rest of this note will be devoted to showing that some of these inclusions become equivalent iff we restrict ourselves to special classes of graphs.

In Section 3 we show that O = SO for complete multipartite graphs. It has been proved that O = C2 for this class [5]. Although the equality O = C2 implies O = SO, we provide

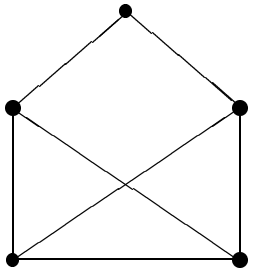

Figure 4: A subgraph-overfull graph which is not neighborhood-overfull.

in Section 3 a simple counting argument.

In Section 4 we show that NO = SO for split graphs. We conjecture that SO = C2 for this class.

In Section 5 we show that NO = SO for indifference graphs. We conjecture that SO = C2 for this class.

We conclude this note with Section 6 where we conjecture that neighborhood-overfullness is the right condition for solving the edge-coloring problem for chordal graphs.

2 Definitions and Notations

In this paper, G denotes a simple, undirected, finite, connected graph. $V(G)$ and $E(G)$ are the vertex and edge sets of G. A *stable set* is a set of vertices pairwise non-adjacent in G. A *clique* is a set of vertices pairwise adjacent in G. A *maximal clique* of G is a clique not properly contained in any other clique. A *subgraph* of G is a graph H with $V(H) \subseteq V(G)$ and $E(H) \subseteq E(G)$. For $X \subseteq V(G)$, we denote by $G[X]$ the *subgraph induced by* X, that is, $V(G[X]) = X$ and $E(G[X])$ consists of those edges of $E(G)$ having both ends in X.

For each vertex v of a graph G, $Adj(v)$ denotes the set of vertices which are adjacent to v. In addition, $N(v)$ denotes the *neighborhood* of v, that is, $N(v) = Adj(v) \cup \{v\}$. A subgraph which is induced by the neighborhood of a vertex is simply called a *neighborhood*. The *degree* of a vertex v is $\deg(v) = |\mathrm{Adj}(v)|$. The maximum degree of a graph G is then $\Delta(G) = \max_{v \in V(G)} \deg(v)$. A vertex u is *universal* if $\deg(u) = |V(G)| - 1$. A k-neighborhood is the neighborhood of a vertex of degree k. For us, K_n denotes the complete graph on $n \geq 1$ vertices.

A vertex v is *simplicial* if $N(v)$ is complete. A *perfect elimination order* of a graph G is a total order on its vertex set $v_1, v_2, \ldots, v_n$ such that for each i, $1 \leq i \leq n$, v_i is simplicial in $G[v_1, v_2, \ldots, v_i]$. A graph is *chordal* if it admits a perfect elimination order.

An *interval graph* is the intersection graph of a set of intervals of the real line. If unitary intervals can be taken, then the graph is called *unitary interval*, *proper interval* or *indifference graph*. We shall adopt the latter name, to be consistent with the terminology of indifference *orders*, defined below. Indifference graphs can be characterized by a linear

order: their vertices can be linearly ordered so that the vertices contained in the same clique are consecutive [10]. We shall call such an order an *indifference order*. By definition, every indifference order is a perfect elimination order.

A *split graph* is a graph whose vertex set admits a partition into a stable set and a clique. Given such a partition for a split graph a perfect elimination order can be defined by placing the vertices in the clique before the vertices in the stable set. So every split graph is a chordal graph.

Let p and $a_1 \leq a_2 \leq \ldots \leq a_p$ be positive integers. The *complete multipartite* graph $K(a_1, \ldots, a_p)$ is defined as follows. It has $n = a_1 + a_2 + \ldots + a_p$ vertices, partitioned into parts $A_1, A_2, \ldots, A_p$ where each A_i has cardinality a_i. If two vertices are in the same part, they are not adjacent, while if they are in different parts, they are adjacent.

3 Complete multipartite graphs

We present in this section a simple counting argument that shows that the particular structure of complete multipartite graphs force subgraph-overfullness to be equivalent to overfullness in this case.

Theorem 2 *Every complete multipartite graph that is subgraph-overfull is overfull.*

Proof: Suppose there exists G, a subgraph-overfull complete multipartite graph that is not an overfull graph itself. Thus by definition, the graph G contains a proper induced subgraph H with same maximum degree such that H is an overfull graph.

Since H is an induced subgraph of G, H is also complete multipartite. Hence H has $n_H = h_1 + h_2 + \ldots + h_p$ vertices, partitioned into stable sets $H_1, H_2, \ldots, H_p$ such that $|H_i| = h_i$ and there is an edge between any two vertices that belong to distinct stable sets. We may assume: $h_1 \leq h_2 \leq \ldots \leq h_p$.

First we note that we must have: $h_1 < h_2 \leq \ldots \leq h_p$. For note that all vertices in H_1 are maximum degree vertices. We have $\Delta(H) = h_2 + \ldots + h_p$. Now because G and H have the same maximum degree any vertex of $G \setminus H$ is missed by all vertices of H_1. If we had $h_1 = h_2$, then we would have $\Delta(G) > \Delta(H)$.

We have $m_H = \sum_{i \neq j} h_i h_j$. On the other hand: $\Delta(H)(n_H - 1)/2 = (h_2 + \ldots + h_p)(h_1 + h_2 + \ldots + h_p - 1)/2$ The overfull condition on graph H says: $m_H > \Delta(H)(n_H - 1)/2$.

This implies the following: $h_1h_2 + h_1h_3 + \ldots + h_1h_p > h_2h_2 + h_3h_3 + \ldots + h_ph_p - (h_2 + h_3 + \ldots + h_p) = h_2(h_2 - 1) + h_3(h_3 - 1) + \ldots + h_p(h_p - 1)$. Now because we have $h_1 < h_2 \leq \ldots \leq h_p$, we have: $h_2(h_2 - 1) + h_3(h_3 - 1) + \ldots + h_p(h_p - 1) \geq h_1h_2 + h_1h_3 + \ldots + h_1h_p$. And this contradiction finishes the proof. ■

4 Split graphs

In this section we prove that for split graphs being subgraph-overfull is equivalent to being neighborhood-overfull.

By definition, every neighborhood-overfull graph is also subgraph-overfull. We show below that for split graphs, every subgraph-overfull graph is neighborhood-overfull. Because the class of split graphs is hereditary, it is enough to show that every overfull split graph is neighborhood-overfull.

We begin by showing that every overfull split graph must contain a universal vertex.

Lemma 9 *If G is split and overfull then G always contains a universal vertex.*

Proof: Suppose that G is a split graph with vertex set partitioned into two sets A and B, such that set A is a clique and set B is a stable set. We may assume that A is a maximal clique.

We use the following notation: $|A| = a$ and $|B| = b$. It follows that: $n = a + b$, $\Delta \geq a$, $m \leq a(a-1)/2 + a(\Delta - a + 1)$.

Let us write $\Delta = a + x$, $x \geq 0$. So each A-vertex sees at most $x + 1$ vertices of B. Each Δ-vertex belongs to set A and sees precisely $x + 1$ B-vertices.

We shall prove that every split graph that is overfull has $|B| = x + 1$. This means that each Δ-vertex belongs to set A and sees every B-vertex. Thus every Δ-vertex is universal.

We have the following condition for a graph to be overfull:

$$m > \Delta(n-1)/2 = (a+x)(a+b-1)/2 = a(a-1)/2 + ab/2 + x(a+b-1)/2.$$

On the other hand, the definition of the partition gives:

$$m \leq a(a-1)/2 + a(\Delta - a + 1) = a(a-1)/2 + a(x+1).$$

These two restrictions on the values of m give:

$$ab/2 + x(a+b-1)/2 < a(x+1).$$

And this in turn implies:

$$b(a+x) - x < ax + 2a.$$

Now suppose for a moment that $b \geq x + 2$. This implies:

$$(x+2)(a+x) - x \leq b(a+x) - x < ax + 2a.$$

And this contradicts $x \geq 0$. ■

Now Lemma 9 implies immediately corollary below.

Corollary 3 *If G is split and subgraph-overfull then G is neighborhood-overfull.*

5 Indifference graphs

In this section we prove that for indifference graphs being subgraph-overfull is equivalent to being neighborhood-overfull.

By definition, every neighborhood-overfull graph is also subgraph-overfull. We show below that for indifference graphs, every subgraph-overfull graph is neighborhood-overfull. Because the class of indifference graphs is hereditary, it is enough to show that every overfull indifference graph is neighborhood-overfull.

We begin by establishing a necessary structural condition for any indifference graph that is overfull. We show that every overfull indifference graph admits a partition of its vertex set into Δ-neighborhoods.

Consider an indifference graph G=(V,E). An indifference order on V associates to each vertex a positive integer i, where $0 \leq i \leq n$. We denote by Δ the maximum degree of G. When we refer to a different graph H, we denote its maximum degree by $\Delta(H)$ accordingly.

Lemma 10 *Suppose that V is not the disjoint union of Δ-neighborhoods. Thus there exists a sequence of vertices $0 = w_0 < u_1 < v_1 < w_1 < ... < u_k < v_k < w_k, (k >= 0)$ satisfying the following properties:*

(i) *all u_i have degree Δ;*

(ii) *$w_i = v_i + 1$, for all $1 \leq i \leq k$;*

(iii) *vertices $w_0, u_1, \ldots, w_k$ in this given order induce a chordless path in G;*

(iv) *there is no vertex x of degree Δ such that $w_k < x$ and $x \in N(w_k) \setminus N(v_k)$.*

Proof: We argue by induction on $n = |V|$. We note that, for $n \leq 3$, G is trivially the disjoint union of Δ-neighborhoods, as G has always a universal vertex. So we may assume that $n \geq 4$.

Suppose that V is not the disjoint union of Δ-neighborhoods and $|V| = 4$. Thus, G has no universal vertex, which implies $\Delta = 2$. In this case, G is isomorphic to a P_4, a path induced by four vertices and those vertices correspondent to a sequence w_0, u_1, v_1, w_1, as required.

For $n > 4$, let u_1 be the rightmost neighbor of $w_0 = 0$. Because $n > 1$, we have $w_0 < u_1$. If $\deg(u_1) < \Delta$, then w_0 is the required sequence.

On the other hand, suppose $\deg(u_1) = \Delta$. Note that $\deg(w_0) = \Delta$ implies that $V = N(w_0)$ is the disjoint union of Δ-neighborhoods. Thus, we have $\deg(u_1) > \deg(w_0)$. Let v_1 be the rightmost neighbor of u_1. We have $u_1 < v_1$. Again, because $V \neq N(u_1)$, there exists $w_1 = v_1 + 1$.

Note that the path induced by w_0, u_1, v_1, w_1 has no chords, by definition of u_1, v_1 as rightmost neighbors of w_0, u_1, respectively.

If there is no vertex x with $\deg(x) = \Delta$, and $x \in N(w_1) \setminus N(v_1)$, then $k = 1$ and w_0, u_1, v_1, w_1 is the required sequence.

On the other hand, in case there is such a vertex x, consider the graph G', induced by w_1 and its successors with respect to the indifference order. By definition, graph G' is itself

indifference with $\Delta(G') = \Delta$. Now if V' is the disjoint union of Δ-neighborhoods, then $V = N(u_1) \cup V'$ and $N(u_1) \cap V' = \emptyset$ imply that V is the disjoint union of Δ-neighborhoods.

Thus by induction for G', we have a sequence $w_1 < u_2 < v_2 < w_2 < ... < u_k < v_k < w_k$ with the required properties. This together with the sequence $w_0 < u_1 < v_1 < w_1$ gives the required sequence for G. Indeed, properties 1, 2 and 4 are trivially satisfied. For property 3, if there was a chord in this path, connecting a vertex before w_1 to a vertex after w_1, we would have the chord v_1u_2. This is not possible, as vertex u_2 has degree Δ in G' and therefore has no neighbors in $N(u_1)$. ■

Lemma 11 *Let G be an indifference graph. Suppose that G admits a sequence $0 = w_0 < u_1 < v_1 < w_1 < ... < u_k < v_k < w_k, (k >= 0)$ defined as above. Then G is not overfull.*

Proof: Let us define a function $c: V \to R$ as follows:

$$\begin{aligned} c(w_i) &= +1, 0 \leq i \leq k, \\ c(v_i) &= -1, 1 \leq i \leq k, \\ c(x) &= 0, \text{otherwise.} \end{aligned}$$

Thus this function satisfies that no neighborhood may contain two non-zero sign vertices with equal signs, as this gives a vertex of degree greater than Δ. Indeed, notice that given w_i, u_{i+1}, v_{i+1}, by definition of this sequence, u_{i+1} is a vertex of degree Δ and its neighbors are precisely those vertices $y \neq u_{i+1}$, such that $w_i \leq y \leq v_{i+1}$.

Thus, there is at most one vertex of degree $+1$ or -1 in each neighborhood $N(x)$, which gives:

$$\sum_{y \in N(x)} c(y) = \begin{cases} 0, & \text{if there exists none or precisely one vertex of each sign in } N(x); \\ +1, & \text{if there exists one } +1 \text{ vertex and no } -1 \text{ vertex in } N(x); \\ -1, & \text{if there exists one } -1 \text{ vertex and no } +1 \text{ vertex in } N(x). \end{cases}$$

Moreover, we note that, for every $x \in V$:

$$\sum_{y \in N(x)} c(y) \leq \Delta - \deg(x)$$

Indeed, we have $\Delta - \deg(x) \geq 0$. Suppose for a moment $\sum_{y \in N(x)} c(y) = +1$ and $\Delta = \deg(x)$. This means x sees a $+1$ vertex and sees no -1 vertex, i.e., x sees w_i but sees no v_j. Thus, $w_i < x$, with $i \neq k$. Hence, $i < k$ and $x < u_{i+1}$ as otherwise x sees v_{i+1}. But now the neighborhood of x lies between w_i and v_{i+1}, and does not contain v_{i+1}. Therefore it is properly contained in $N(u_{i+1})$, which contradicts $\deg(x) = \Delta$.

Now,

$$\sum_{x \in V} (1 - c(x))(\Delta - \deg(x) - \sum_{y \in N(x)} c(y)) \geq 0,$$

as it is a sum of positive terms. This in turn gives,

$$\sum_{x \in V} \Delta - \sum_{x \in V} \deg(x) - \sum_{x \in V} \sum_{y \in N(x)} c(y) - \sum_{x \in V} c(x)\Delta + \sum_{x \in V} c(x) \deg(x) + \sum_{x \in V} c(x) \sum_{y \in N(x)} c(y) \geq 0$$

Note that $\sum_{x \in V} c(x) = 1$, as w_0 is a $+1$ vertex and all other w_i cancel with a v_i. Hence,

$$n\Delta - 2m - \sum_{x \in V} \sum_{y \in N(x)} c(y) - \Delta + \sum_{x \in V} c(x) \deg(x) + \sum_{x \in V} (c(x))^2 + 2 \sum_{xy \in E} c(x)c(y) \geq 0$$

i.e.,

$$n\Delta - 2m - \sum_{y \in V} (c(y)(1 + \deg(y))) - \Delta + \sum_{x \in V} c(x) \deg(x) + (2k + 1) - 2k \geq 0$$

and finally,

$$n\Delta - 2m - \Delta \geq 0$$

i.e., $(n-1)\Delta \geq 2m$, which implies that the graph is not overfull. ■

Theorem 3 *If $G = (V, E)$ is indifference and overfull then V can be partitioned into Δ-neighborhoods. In particular, we have $n = k(\Delta + 1)$, for some integer $k \geq 1$.*

Proof: The result is a direct consequence of Lemma 10 and Lemma 11. ■

We are now ready to establish that for indifference graphs subgraph-overfullness is equivalent to neighborhood-overfullness. By definition, every neighborhood-overfull graph is subgraph-overfull. The following theorem proves the converse.

Theorem 4 *If G is indifference and subgraph-overfull then G is neighborhood-overfull.*

Proof: It suffices to prove that if G is indifference and overfull then G is neighborhood-overfull.

Let $G = (V, E)$ be an indifference overfull graph. In view of Theorem 3, we know that $n = k(\Delta + 1)$. Furthermore, since n must be odd, we have k odd and Δ even. We will assume further that G is *not* neighborhood-overfull and arrive at a contradiction.

Consider the vertices of G in an indifference order. Theorem 3 allows us to divide the vertex set V into k consecutive blocks of $\Delta + 1$ vertices each. Let $V_1, V_2, \ldots, V_k$ be these blocks.

Our goal is to count the edges and show that G cannot be overfull.

Let A_i be the set of edges whose left end point is in V_i. We claim that

$$|A_i| \leq \frac{\Delta(\Delta + 1)}{2}, \text{ for } 1 \leq i \leq k - 1, \tag{1}$$

and

$$|A_k| \leq \frac{\Delta(\Delta + 1)}{2} - \frac{\Delta}{2}. \tag{2}$$

Notice that Claim (2) is immediate from our assumption that G is not neighborhood-overfull, since A_k is the edge set of the neighborhood $G[V_k]$. Let's prove Claim (1).

There are two kinds of edges in A_i, for $1 \leq i \leq k - 1$: those whose right end point is in V_i and those that go past V_i. The crucial fact here is that, for each edge going past V_i, there

is an edge missing between vertices of V_i. Indeed, let $u \in V_i$ be the left end point of an edge going past V_i. Since $\deg(u) \leq \Delta$, for each vertex that u sees beyond V_i there must be a vertex in the beginning of V_i that u does not see. (This is because the closed neighborhood $N[u]$ contains at most $\Delta + 1$ consecutive vertices.) Therefore, the total number of edges in A_i is bounded by the number of edges in a complete graph over V_i, that is, $\Delta(\Delta + 1)/2$.

Now let's use (1) and (2). Notice that every edge of G must start somewhere, hence

$$\begin{aligned} m &= |E| \\ &= \sum_{i=1}^{k} |A_i| \\ &\leq \sum_{i=1}^{k-1} \frac{\Delta(\Delta+1)}{2} + \frac{\Delta(\Delta+1)}{2} - \frac{\Delta}{2} \\ &= k\frac{\Delta(\Delta+1)}{2} - \frac{\Delta}{2} \\ &= \frac{\Delta}{2}(n-1), \end{aligned}$$

showing that G cannot be overfull. This contradiction concludes our proof. ■

6 Conclusions

We have the following conjecture about edge-coloring chordal graphs:

Conjecture 1 *Every C2 chordal graph is neighborhood-overfull.*

We are currently working on two necessary conditions for this conjecture. Firstly, we need all chordal graphs that are subgraph-overfull to be also neighborhood-overfull. In this paper, we have established this fact for two subclasses of chordal graphs: split graphs and indifference graphs.

Secondly, we need all odd maximum degree chordal graphs to be C1. We have established this fact for indifference graphs [2].

Let us consider the class of complete multipartite graphs. It has been shown recently that any C2 complete multipartite graph is overfull [5]. This implies in particular that for complete multipartite graphs being overfull is equivalent to being subgraph-overfull. In this paper, we have shown a simple counting argument that provides an alternate proof for that fact. Now consider the complete multipartite graph with 9 vertices with parts: A_1, A_2, A_3, where each part has 3 vertices. This graph is overfull but is not neighborhood-overfull. Now consider the complete multipartite graph with 7 vertices with parts: A_1, A_2, A_3, where $a_1 = a_2 = 2$ and $a_3 = 3$. This graph is overfull but is not neighborhood-overfull, as its maximum degree is odd.

We have examples of odd maximum degree graphs that are overfull. All such examples are not chordal graphs. We conjecture that an odd maximum degree chordal graph cannot be overfull.

One way to prove that an overfull graph is neighborhood-overfull is to exhibit a universal vertex. That is what we did for the case of split graphs.

Now consider the graph H obtained by removing an edge ab from K_7. This graph is neighborhood-overfull. Now consider the graph F obtained as follows. Take three copies of H, say H_1, H_2, H_3, where $H_i = K_7 \setminus a_i b_i$ respectively. Add edges $b_1 a_2$ and $b_2 a_3$. This graph is indifference, it is overfull, it is neighborhood-overfull but it contains no universal vertex.

References

[1] L. Cai and J. A. Ellis. NP-completeness of edge-colouring some restricted graphs. *Discrete Applied Mathematics*, 30:15–27, 1991.

[2] C. M. H. de Figueiredo, J. Meidanis, and C. P. de Mello. On edge-colouring indifference graphs. *Lecture Notes in Computer Science*, 911:286–299, 1995.

[3] A. J. W. Hilton. Two conjectures on edge-colouring. *Discrete Math.*, 74:61–64, 1989.

[4] A. J. W. Hilton and P. D. Johnson. Graphs which are vertex-critical with respect to the edge-chromatic number. *Math. Proc. Camb. Phil. Soc.*, 102:211–222, 1987.

[5] D. G. Hoffman and C. A. Rodger. The chromatic index of complete multipartite graphs. *J. of Graph Theory*, 16:159–163, 1992.

[6] I. Holyer. The NP-completeness of edge-coloring. *SIAM J. Comput.*, 10:718–720, 1981.

[7] J. Misra and D. Gries. A constructive proof of vizing's theorem. *Inf. Proc. Lett.*, 41:131–133, 1992.

[8] M. Plantholt. The chromatic index of graphs with a spanning star. *J. of Graph Theory*, 5:45–53, 1981.

[9] M. Plantholt. The chromatic index of graphs with large maximum degree. *Discrete Math.*, 47:91–96, 1983.

[10] F. S. Roberts. On the compatibility between a graph and a simple order. *J. Comb. Theory (B)*, 11:28–38, 1971.

[11] V. G. Vizing. On an estimate of the chromatic class of a p-graph. *Diket. Analiz.*, 3:25–30, 1964. In russian.

Uma Adaptação do Algoritmo de Emparelhamento de Edmonds para Execução em Paralelo

Carlos Fernando Bella Cruz*
João Carlos Setubal*
Departamento de Ciência da Computação
IMECC–UNICAMP
Caixa Postal 6176
13081-970Campinas, SP
{cfbcruz,setubal}@dcc.unicamp.br

Abstract: Apresentamos uma adaptação do algoritmo de emparelhamento de Edmonds para execução em paralelo. Nosso objetivo é execução eficiente na prática. A adaptação consiste em permitir que cada processador procure caminhos aumentantes no grafo de forma independente dos demais. Embora a busca ocorra de forma paralela, o aumento do emparelhamento é feito apenas por 1 processador de cada vez, o que garante a corretude do algoritmo sem incorrer em atraso significativo no tempo de execução. Resultados preliminares de uma implementação indicam que a adaptação proposta consegue ser mais rápida do que uma boa implementação seqüencial em alguns tipos de grafos. Esses testes foram feitos num multiprocessador de memória compartilhada com 8 processadores. O desenvolvimento da implementação teve como antecendente uma experiência negativa de paralelização do algoritmo de Micali-Vazirani.

1 Introdução

Seja $G = (V, E)$ um grafo não orientado com n vértices e m arestas. Um *emparelhamento* M é um subconjunto das arestas tal que não existem duas arestas de M que incidem no mesmo vértice. Um emparelhamento é *máximo* se sua cardinalidade for máxima.

Nosso objetivo de pesquisas é encontrar um algoritmo paralelo para resolver o problema do emparelhamento máximo que seja eficiente *na prática*. Por prática entendemos execução eficiente num multiprocessador de memória compartilhada com poucos (< 20) processadores. Neste trabalho apresentamos uma adaptação simples do algoritmo de Edmonds [Edm65] para execução em paralelo com a qual obtivemos resultados preliminares encorajadores.

O trabalho é apresentado da seguinte forma. Na seção 2 apresentamos um breve histórico dos algoritmos para o problema do emparelhamento. Na seção 3 descrevemos uma primeira tentativa de obtenção de uma implementação baseada no algoritmo de Micali e Vazirani [MV80]. Na seção 4 apresentamos uma descrição detalhada do algoritmo proposto, e na seção 5 apresentamos nossos resultados computacionais preliminares. O artigo é concluído na seção 6.

*Este trabalho é financiado, em parte, por auxílios do CNPq e da FAPESP.

2 Breve Histórico

Para apresentar este rápido histórico dos algoritmos para o problema do emparelhamento, e também para o restante do artigo, precisamos da noção de *caminho aumentante*.

Dado um emparelhamento M, um vértice é considerado *livre* se nenhuma das arestas nele incidentes pertence a M. Um *caminho alternado* é um caminho simples cujas arestas se alternam entre M e $E-M$. Um *caminho aumentante* é um caminho alternado entre dois vértices livres. A partir de um caminho aumentante P podemos obter um emparelhamento M' retirando de M as arestas pertencentes a P e incluindo em M as arestas restantes. O caminho se chama aumentante porque $|M'| = |M| + 1$.

Berge [Ber57] e, independentemente Norman e Rabin [NR59], mostraram que um emparelhamento é máximo se e somente se não admite caminhos aumentantes. O primeiro algoritmo polinomial ($O(n^4)$) foi proposto por Edmonds [Edm65]. Este resultado foi progressivamente melhorado. Gabow [Gab76] obteve uma implementação do algoritmo de Edmonds com tempo $O(n^3)$. Even e Kariv [EK75] obtiveram um algoritmo com tempo $O(n^{\frac{5}{2}})$. Micali e Vazirani [MV80] obtiveram um algoritmo com tempo $O(\sqrt{n}m)$. Atualmente, este é o algoritmo mais rápido conhecido. Blum [Blu94] apresentou um novo enfoque para o algoritmo de Micali e Vazirani, porém com a mesma complexidade de tempo.

Na computação paralela, Mulmuley, Vazirani e Vazirani (MVV) [MVV87] apresentaram um algoritmo randomizado para o modelo PRAM baseado em inversão de matrizes, com tempo de execução $O(\log^2 n)$. Este algoritmo infelizmente parece pouco prático. Em primeiro lugar, sabe-se que o modelo PRAM é distante das máquinas paralelas atualmente existentes. Em segundo lugar, o número de operações do algoritmo de MVV é cerca de $O(n^{3.5})$, muito superior ao número de operações do algoritmo de Micali e Vazirani, que é $O(n^{1.5})$ no caso de grafos esparsos (de maior interesse na prática). Além disso as constantes embutidas no tempo de execução do algoritmo de MVV são altas, em particular pelo uso da rotina para inversão de matrizes.

3 Tentativa de Paralelização do Algoritmo de Micali e Vazirani

Com as considerações feitas na seção anterior, para nós tornou-se claro que uma implementação paralela eficiente na prática deveria se basear em algum dos algoritmos seqüenciais existentes. Dentre esses, o mais promissor nos pareceu o de Micali e Vazirani, pelas razões expostas a seguir.

O algoritmo de Micali e Vazirani utiliza o conceito de *fases*. Em cada fase são encontrados todos os caminhos aumentantes disjuntos de menor comprimento. Hopcroft e Karp [HK73] provaram que são necessárias $\sqrt{n}$ fases para se encontrar o emparelhamento máximo. Micali e Vazirani mostraram como realizar cada fase em tempo $O(m)$. A complexidade resulta portanto $O(\sqrt{n}m)$.

Cada fase é composta de buscas em largura a partir de todos os vértices livres, e buscas em profundidade para "juntar" as buscas entre si e trocar as arestas dos caminhos encontrados. Nosso objetivo na implementação foi de realizar em paralelo tanto as bus-

cas em largura quanto as buscas em profundidade, e realizando sincronização apenas entre uma fase e a seguinte. Entretanto, embora o algoritmo possa ser descrito dessa forma aparentemente simples em termos de "buscas", na verdade cada busca precisa realizar uma rotulagem cuidadosa dos vértices e das arestas. Além disso, para manter as características do algoritmo original é necessário realizar sincronizações a cada nível da busca em largura. Finalmente, a pequena da granularidade das tarefas (de modo que vários processadores possam colaborar numa determinada busca) acaba por gerar uma aumento inaceitável de operações na execução. Um protótipo de implementação paralela foi construído e todos esses problemas foram constatados na prática. Diante disso, decidimos partir para uma outra abordagem, conforme descrito na próxima seção.

4 Uma Adaptação do Algoritmo de Edmonds

Após uma análise dos problemas encontrados na implementação paralela baseada no algoritmo de Micali e Vazirani, constatamos que uma alternativa viável seria, não uma cooperação dos processadores para a execução de um algoritmo, mas sim que cada um executasse um algoritmo independente, com seu próprios dados. Esta nova abordagem será explicada em seguida.

4.1 Visão Geral

A idéia básica da adaptação é simplesmente permitir que os processadores procurem caminhos aumentantes de forma independente e assíncrona (ou seja: cada processador executa por si próprio o algoritmo de Edmonds). Toda vez que um processador encontra um caminho aumentante a estrutura de dados que armazena o emparelhamento é atualizada. Esta atualização é feita de forma exclusiva (somente um processador por vez).

A estrutura de dados que armazena o emparelhamento é simplesmente um vetor de vértices, o qual chamamos C. Se os vértices u e v estão emparelhados, então $C(u) = v$ e $C(v) = u$.

A primeira tarefa do algoritmo consiste em dividir uniformemente os vértices entre os processadores. Cada processador p_i analisa seus vértices e constroi uma lista ligada l_i, aonde cada elemento armazena um vértice livre identificado. As listas l_i de cada processador são unidas numa lista global L, assim que forem sendo terminadas.

A tarefa de um processador p_i consiste em adquirir um vértice livre da lista L. Isto é feito numa região de exclusão mútua para garantir que somente um único processador esteja realizando uma busca a partir de um vértice livre. Por outro lado, mais de um processador pode estar realizando sua respectiva busca ao longo das mesmas arestas. Isso é possível pois as informações de cada busca são exclusivas de cada processador.

Quando um processador encontra um caminho aumentante, ele procura realizar a troca das arestas. Porém, somente um processador pode estar realizando a troca num dado instante. Esta exclusividade é necessária para garantir que o emparelhamento encontrado seja máximo. Pelo fato da execução ser em paralelo, é possível que um processador encontre inconsistências no grafo tanto durante a busca quanto na troca de arestas de um caminho

aumentante. Nesse caso o processador refaz as buscas. Na seção 4.4 abaixo apresentamos uma demonstração de que com estas características o algoritmo está correto.

Uma vez que nosso objetivo é prático, a implementação incorpora uma rotina inicial que busca um emparelhamento maximal simples, onde se procura, em paralelo, emparelhar cada vértice com um de seus vizinhos. Testes empíricos mostraram que esta rotina inicial já alcança pelo menos 80% do emparelhamento máximo (o que não é surpresa, visto que qualquer emparelhamento maximal alcança pelo menos 50% do máximo). Resta à segunda parte da implementação tentar emparelhar os restantes 20%.

4.2 Detalhamento da Implementação

Nesta seção detalhamos aspectos relativos ao processo de busca de caminhos aumentantes feito por um processador. Esta descrição supõe que a implementação está sendo executada seqüencialmente, e portanto é essencialmente uma descrição do algoritmo de Edmonds, com algumas modificações obtidas a partir do algoritmo de Micali e Vazirani. Para este último nos baseamos na descrição de Peterson e Louie [PL88]. Na seção seguinte descreveremos os problemas e as mudanças necessárias quando a execução ocorre em paralelo.

A implementação possui três subrotinas principais (além da rotina de emparelhamento inicial), que são: *procurar_caminho*, *construir_flor* e *identificar_caminho*. A subrotina *procurar_caminho* é responsável pela expansão da busca no grafo. A expansão da busca é feita até que seja encontrado um caminho aumentante ou até que todos os vértices sejam visitados. Durante a expansão, a subrotina poderá identificar algumas estruturas chamadas de *flores*. Esta estrutura possui características bem definidas que serão descritas posteriormente. Cada vez que uma flor é identificada, a subrotina *construir_flor* é chamada para tratá-la adequadamente. No final da expansão da busca, se um caminho aumentante for encontrado a subrotina *identificar_caminho* é chamada para identificar as arestas que o compõem e aumentar o emparelhamento.

A expansão da busca a partir de um vértice livre r é feita de maneira idêntica ao algoritmo de Edmonds. O tratamento dado às flores foi baseado no algoritmo de Micali e Vazirani. Embora este algoritmo não utilize o conceito de flor como é definido no algoritmo de Edmonds, os seus conceitos foram adaptados para a nova situação.

Para a subrotina *procurar_caminho* controlar a expansão da busca, ela mantém uma lista dos vértices que pertencem à fronteira da árvore de busca. Os vértices nesta lista são rotulados *par* ou *ímpar*, e um vértice v não poderá ser inserido na lista se ele já estiver presente nela. Inicialmente, o único vértice pertencente à lista é o vértice livre r, o qual é marcado como *visitado*. Ele será a raiz da busca e possui rótulo par.

A subrotina *procurar_caminho* inicia a busca retirando um vértice v da lista. Se o vértice v possui rótulo par então para cada vértice u adjacente a v tal que a aresta (u, v) não está emparelhada é feita a seguinte verificação. Se o vértice $u \neq r$ é um vértice livre então foi identificado um caminho aumentante. No entanto, se o vértice u estiver emparelhado, então a busca deve ser expandida para ele. Isto é feito marcando o vértice v como *predecessor* de u, rotulando u com ímpar e visitado e inserindo-o na lista de vértices rotulados. Agora, se o vértice v, retirado da lista, possui rótulo ímpar, então a aresta emparelhada (v, u) é analisada e a única ação a ser feita é expandir a busca. Isto é feito marcando o vértice

v como predecessor de u, rotulando u com par e visitado e inserindo-o na lista de vértice rotulados. A busca prossegue desta maneira até encontrar algum caminho aumentante ou até acabarem os vértices da lista.

Como foi descrito anteriormente, durante a expansão da busca, é possível encontrar estruturas chamadas de flor. Por isto, a cada expansão, a subrotina *procurar_caminho* verifica se é possível atribuir um rótulo par a um vértice visitado com rótulo ímpar ou atribuir um rótulo ímpar a um vértice visitado com rótulo par. Esta é a condição para a ocorrência de uma flor. As características desta estrutura serão descritas em mais detalhes logo abaixo.

Uma flor é basicamente um circuito de comprimento ímpar conectado à uma estrutura chamada *haste*. Formalmente:

Definição 1 *Uma* haste *é um caminho de comprimento par que parte da raiz da busca r e vai até um vértice w. É permitido que r seja igual a w, quando dizemos que a haste é vazia.*

Definição 2 *Uma flor é um circuito de comprimento ímpar que começa e termina no vértice w da haste e não tem nenhum vértice em comum com ela, além de w. O vértice w é a base da flor.*

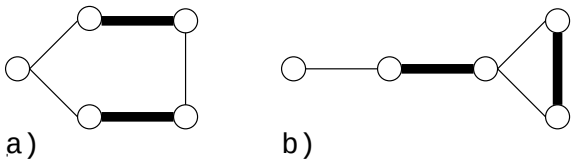

Figura 1: a) Flor com haste vazia. b) Flor com haste.

A figura 1a representa uma flor com haste vazia e a figura 1b representa uma flor com haste não vazia.

Uma flor e sua haste possuem duas propriedades importantes:

1. Uma haste possui $2i + 1$ vértices e contém i arestas emparelhadas para $i \geq 0$; uma flor possui $2j + 1$ vértices e contém j arestas emparelhadas para $j \geq 1$ e as arestas emparelhadas emparelham todos os vértices da flor exceto a base, a qual é um vértice com rótulo par.

2. Pelo fato de uma flor ser um circuito ímpar, todos os seus vértices alcançam a haste por dois caminhos, um deles de comprimento par e outro de comprimento ímpar.

Estas propriedades nos permitem tirar as seguintes conclusões: as arestas que não pertencem a uma flor, mas que incidem num de seus vértices são arestas não emparelhadas e os caminhos que passam por alguma destas arestas possuem um caminho até a base. Portanto, uma flor não constitui um obstáculo para um caminho que passe através dela, o que conduz à idéia de contrair a flor em um pseudo-vértice sobre a sua base [Edm65]. As

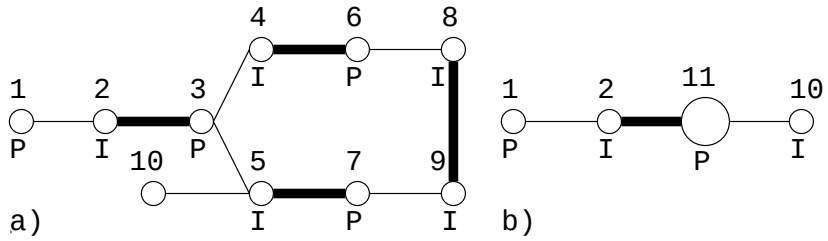

Figura 2: a) Identificação de uma flor. b) Contração de uma flor sobre a sua base

arestas que antes eram incidentes sobre os vértices da flor agora são incidentes sobre esse pseudo-vértice.

Na figura 2a observamos a identificação e contração de uma flor feita pelo algoritmo de Edmonds (Os vértices pares estão marcados com a letra P e os ímpares com a letra I). A raiz da árvore de busca é o vértice 1. Um ramo da busca segue pelos vértices $(1, 2, 3, 4, 6, 8)$ e outro por $(1, 2, 3, 5, 7, 9)$. O vértice 8 pode receber tanto rótulo par quanto ímpar. Isto caracteriza a presença de uma flor. Os vértices que a compõem são: $(3, 4, 5, 6, 7, 8, 9)$. A sua base é o vértice 3 e a haste é o caminho $(1, 2, 3)$. A flor é contraída sobre a sua base, criando-se um pseudo-vértice 11. O resultado da contração está representado na figura 2b.

O tratamento dado à flor pela subrotina *construir_flor* é diferente do tratamento dado pelo algoritmo de Edmonds. A partir das definições acima, pode-se provar que contrair uma flor num pseudo-vértice é equivalente a rotular todos os vértices da flor com rótulo par. Pois deste modo permitiremos o prosseguimento da busca pelas arestas fora da flor. Este é o processo utilizado pela função *construir_flor*. Os vértices da flor com rótulo ímpar recebem rótulo par e inseridos novamente na lista de vértices com rótulos, para serem reavaliados.

Para a identificação dos vértices da flor e para a rotulagem par dos vértices com rótulo ímpar, é necessário que haja uma numeração dos vértices em ordem crescente a partir da raiz da árvore de busca. A cada expansão da busca, o vértice é numerado com o valor do predecessor acrescentado de 1. É possível provar que na formação de uma flor, a base possui o menor valor entre os seus vértices. A explicação de como a subrotina *construir_flor* identifica os vértices da flor (processo baseado no algoritmo de Micali e Vazirani) será feita utilizando o exemplo abaixo.

Na figura 2a, foi identificada uma flor a partir da análise do vértice 9. Definimos a aresta $(9, 8)$ como sendo uma *ponte* e os vértices 8 e 9 como sendo os *picos* da flor. Eles estão numerados com o valor 5. Um pico é rotulado com *direito* e outro com *esquerdo*. A partir destes dois vértices são disparadas duas buscas em profundidade, uma direita e outra esquerda, seguindo as marcações de predecessores. Cada vértice visitado por elas é rotulado como pertencente à busca direita ou esquerda. Elas encontrarão o primeiro vértice em comum entre elas. Este vértice é a base da flor. As buscas são sincronizadas pelos valores contidos nos vértices e a busca esquerda tem prioridade sobre a direita, caso as elas estejam sobre vértices com o mesmo valor. O vértice onde ocorrer o encontro entre as buscas é a base da flor e os arcos percorridos são os vértices pertencentes a ela. No caso da figura 2a, elas encontrarão o vértice 3, que possui valor 2. Se o pico 8 for o ponto de partida da busca esquerda os vértices 6 e 4 serão marcados como esquerdos, o mesmo para os vértices 7 e

5 em relação ao pico 9. Os vértices com rótulos ímpares, 4, 5, 8 e 9, são marcados como pares e inseridos novamente na lista de vértices rotulados. O vértice 5, quando reavaliado, marcará o vértice 10 como ímpar e identificará a presença de um caminho aumentante.

No algoritmo de Edmonds, quando um caminho aumentante é identificado, as suas arestas são analisadas e se houver algum pseudo-vértice, ele será expandido para a sua forma anterior. Em seguida, é verificado se o caminho até a base da flor percorre o arco de comprimento par ou de comprimento ímpar. Na figura 2b, foi encontrado o caminho aumentante $(1, 2, 11, 10)$. Como existe um pseudo-vértice, ele é expandido e se constata que o caminho vém do vértice 10, percorre a flor pelo caminho de comprimento par $(5, 7, 9, 8, 6, 4, 3)$ até a base e segue pela haste $(3, 2, 1)$. É importante notar que é possível ocorrer flores dentro de flores.

A identificação do caminho aumentante feita pela subrotina *identificar_caminho* é feita verificando se existe algum vértice que pertence a uma flor. Se existir então ele é analisado para indicar por onde o caminho continua. Este processo será explicado abaixo com mais detalhes.

Quando um caminho aumentante é identificado a subrotina *identificar_caminho* é chamada com dois vértices como parâmetros. Um deles é o vértice livre recém descoberto e o outro é a raiz da árvore de busca r. A subrotina *identificar_caminho* é responsável pela construção do caminho entre estes dois vértices. Para auxiliar neste processo, os vértices são rotulados como *externos* e *internos* durante a contrução da busca. Um vértice é rotulado interno (externo) se o caminho entre ele e a raiz da busca, seguindo as marcas de predecessores, tem comprimento ímpar (par). O objetivo da subrotina é partir do vértice alto e, seguindo as marcações de predecessores, encontrar o vértice baixo. Se, durante a descida, ele encontrar um vértice v pertencente a uma flor F, a subrotina *identificar_caminho* chama uma pequena subrotina chamada *abre_flor* e passa como parâmetro o vértice v pertencente à flor.

A subrotina *abre_flor* é responsável pela identificação do caminho através de uma flor. Para isto, ela, ao receber o vértice v como parâmetro, verifica se ele é externo ou interno. Se for externo então o caminho parte de v e segue até a base da flor F. Neste caso a subrotina *abre_flor* chama a subrotina *identificar_caminho* recursivamente com os parâmetros v e $base(F)$ respectivamente. Se for interno, o caminho passa através dos picos da flor. Neste caso, se o vértice v possui rótulo direito (esquerdo) então a subrotina *abre_flor* chama a subrotina *identificar_caminho* duas vezes, uma para encontra o caminho entre o pico direito (esquerdo) da flor F e o vértice v e outra para encontra o caminho entre o pico esquerdo (direito) e $base(F)$, contornando a flor F.

4.3 Execução em Paralelo

Conforme mencionado, a atualização do emparelhamento corrente após a descoberta de um caminho aumentante só pode ser feita por um processador. A seguir descrevemos a razão dessa exclusividade, mostrando o que pode ocorrer se ela não existir.

Na figura 3 observamos um caso em que o emparelhamento pode não crescer. Os três caminhos aumentantes são identificados ao mesmo tempo e as atualizações podem seguir a seguinte ordem: num primeiro passo, $C(1) = 2$, $C(2) = 3$ e $C(3) = 1$; num segundo passo, $C(1) = 3$, $C(2) = 1$ e $C(3) = 2$. Portanto, nenhuma aresta foi emparelhada.

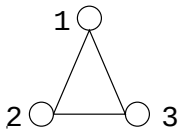

Figura 3: Caso de concorrência entre os processadores.

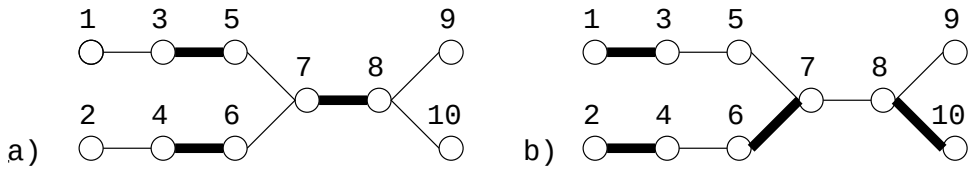

Figura 4: a) Identificação de dois caminhos. b) Troca de vértice livre.

Quando o algoritmo é executado em paralelo, é possível que um vértice que não era livre passe a sê-lo (dizemos que houve uma *troca*). Na figura 4, observamos que somente é possível tratar um caminho aumentante. Porém, é possível que dois processadores identifiquem dois caminhos distintos, um entre os vértices 1 e 9 e outro entre 2 e 1, e que tentem tratá-los ao mesmo tempo. Quando ocorrerem as alterações no emparelhamento temos $C(5) = 7$ e $C(6) = 7$. A atualização de $C(7)$ definirá qual aresta estará emparelhada. Isto é feito pelo último processador a atualizar o campo. Supondo que $C(7) = 6$, observamos o vértice 5 torna-se livre.

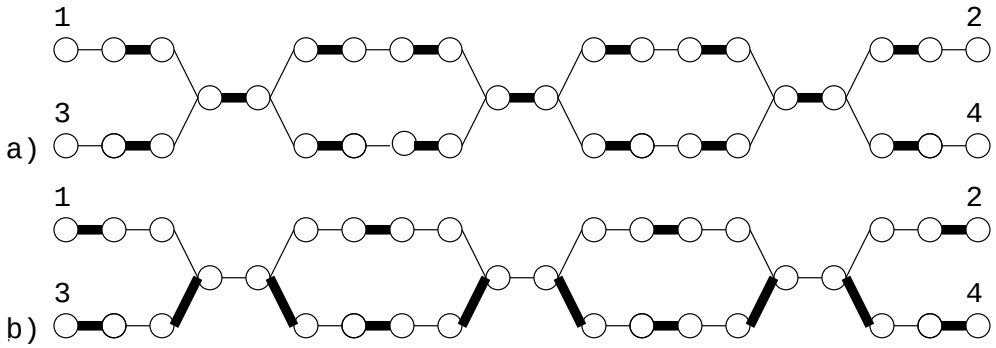

Figura 5: a) Identificação de dois caminhos. b) Após a atualização.

Além de "troca" de vértices livres, é possível haver a criação de novos vértices livres. Na figura 5, quando dois processadores atualizam os dois caminhos de 1 a 2 (passando pelos arcos superiores) e de 3 a 4 (passando pelos arcos inferiores) ao mesmo tempo, ocorre a criação de dois vértices livres. Antes tínhamos 15 arestas emparelhadas e depois da atualização obtemos somente 14, como mostra a figura 5b, ocorrendo, portanto, uma diminuição do emparelhamento. Além do mais é possível haver a criação de vértices livres que estejam adjacentes. Neste caso é possível que o emparelhamento gerado não seja sequer maximal.

Estes problemas são solucionados aplicando-se uma área de exclusão mútua quando um caminho aumentantes é identificado. Deste modo somente um processador pode atualizar

o emparelhamento num determinado instante. Entretanto, os outros processadores continuam executando as buscas. Isto faz com que haja uma interferência da atualização do emparelhamento com as buscas dos outros processadores. Esta interferência faz com que seja necessário um tratamento especial por parte do algoritmo.

Devido às constantes mudanças do emparelhamento, devemos tomar cuidado nas expansões das buscas. Não é possível mais confiar nas informações das arestas emparelhadas. Anteriormente definimos que quando se expande uma busca a partir de um vértice v com rótulo par, deve-se analisar todos os vértices adjacentes u, cujas arestas (v, u) não estejam emparelhadas. Num algoritmo seqüencial, a aresta emparelhada representa o caminho de onde veio a busca, mas em paralelo isto pode não ser verdade. Devemos, portanto, analisar todos os vértices u cujas arestas (v, u) não estejam emparelhadas e tal que o vértice u não esteja marcado como predecessor de v. Quando o vértice v tem rótulo ímpar deve-se expandir a busca para a aresta emparelhada (v, u) somente se o vértice u não está marcado como predecessor de v. Esta descrição nos permite formular a seguinte definição:

Definição 3 *Uma* inconsistência *numa busca ocorre quando um processador supõe que uma determinada aresta esteja emparelhada e ela não está, e vice-versa.*

Um processador pode detectar uma inconsistência em 2 momentos: durante uma busca, ou durante a efetivação de um caminho aumentante. Para lidar com as inconsistências adotamos a solução simples de obrigar o processador a refazer a busca a partir do vértice livre original. Com esta solução, o único problema que ainda poderia haver seria no caso de um processador não encontrar um caminho aumentante e nem inconsistências. Na próxima seção mostramos que esse caso não causa problemas, e portanto o algoritmo sempre acha um emparelhamento máximo.

A aplicação da área de exclusão mútua para aumento do emparelhamento e a necessidade de refazer certas buscas não produziram atrasos significativos em nossos testes. Segundo nossas medições, o trabalho necessário para tratar um caminho aumentante é muito menor do que o trabalho necessário para tratar flores e as expansões das buscas, e o número de buscas que precisam ser refeitas também é pequeno. Entretanto não é difícil imaginar grafos cuja estrutura provocaria um péssimo desempenho do algoritmo.

As informações necessárias num processador são: o predecessor de cada vértice, o valor atribuído durante a expansão da busca, as informações booleanas (par, ímpar, interno, externo, direita, esquerda, visitado e não visitado) e a identificação da flor a qual cada vértice pode pertencer. Cada flor é identificada por um elemento de uma lista ligada, onde são armazenados os picos e a base da flor. A memória necessária para cada processador é de $O(n)$. Portanto, a execução do algoritmo ocupa um espaço total de $O(pn)$, onde p é o número de processadores utilizados.

4.4 Demonstração de Corretude

Nesta seção demonstramos que o algoritmo proposto sempre encontra um emparelhamento máximo. Considerando que o que cada processador faz é executar sua "cópia" do algoritmo de Edmonds, e que cada emparelhamento é alterado de forma exclusiva, o seguinte lema é suficiente para demonstrar a corretude.

Lema 1 *Se um processador não consegue encontrar um caminho aumentante a partir de um vértice v, e tampouco encontra inconsistências durante a busca, um tal caminho não existe, mesmo levando-se em conta que o emparelhamento pode estar sendo alterado por outros processadores durante a busca.*

Para provar esse lema precisamos da seguinte definição:

Definição 4 *Dado um vértice livre v e a árvore de busca A a partir de v, chamamos de* vértices de fronteira *de A todo vizinho de um vértice ímpar de A (após todas as contrações de flor que se fizerem necessárias).*

É fácil ver que vértices de fronteira não pertencem a A (isto é, não são visitados a partir de v).

Suponha agora que o processador i não encontrou um caminho aumentante a partir de v e nem inconsistências, mas um tal caminho existe. Considere a árvore de busca A a partir de v. Se o caminho existe, ele necessariamente chega em v através de algum vértice de fronteira de A, digamos f. Considere agora a situação vigente durante a formação de A. A aresta que une f a $u \in A$ era uma aresta não emparelhada quando u foi visitado. Portanto, a busca prosseguiu pela aresta emparelhada incidente em u, digamos (u, w). Considere agora a descoberta do caminho aumentante e sua conseqüente alteração por um outro processador j. A aresta (f, u) pertence ao caminho, por hipótese, e será alterada de não-emparelhada para emparelhada. A aresta seguinte do caminho deverá ser (u, w), e mudará de emparelhada para não-emparelhada. Vemos portanto que a alteração do caminho e a busca que forma A *seguem no mesmo sentido.* Agora duas situações podem ocorrer: ou as alterações ficam sempre "atrás" da busca, ou "ultrapassam" a busca. Se houver ultrapassagem, necessariamente a busca revelará uma inconsistência, e essa possibilidade está excluída por hipótese. Se as alterações permanecerem atrás, então a própria busca encontrará de novo o vértice v, o que vale dizer que v pertence a uma flor. Mas esta situação é impossível pelo fato de f ser um vértice de fronteira. Portanto tal caminho não existe e o lema está provado.

5 Resultados Experimentais Preliminares

5.1 A Máquina

A máquina utilizada é uma Sparc Server 1000 com 8 processadores, com sistema operacional Solaris. A implementacão paralela foi feita através de *threads* (novos pontos de execução sobre o mesmo código). É possível criar tantos *threads* quantos necessários. Para tentar minimizar as influências relacionadas com escalonamento de *threads*, a implementação procura fazer com que os *threads* cumpram o papel de processadores físicos, associando 1 *thread* a cada processador e mantendo-os durante todo o programa. Infelizmente o sistema operacional impede que esta vinculação física se realize, e portanto no decorrer da execução 1 *thread* pode ser re-escalonado para outro processador.

Para medir o tempo gasto pela implementação paralela, foram usadas duas funções: `gethrtime` e `gethrvtime`. Elas possuem características bem distintas, sendo necessário analisá-las para se obter resultados confiáveis.

A função `gethrtime` mede o tempo de relógio. Portanto, o tempo medido inclui o tempo gasto no escalonamento dos *threads* e o tempo gasto nas áreas de exclusão mútua. Já a função `gethrvtime` não inclui nenhum dos dois, pois ela mede somente o tempo de execução do código. Isto significa que essas rotinas nos dão um limite inferior e superior para o real tempo de execução da implementação. Assim sendo, apresentamos nossos tempos na forma de um intervalo. Em geral esse intervalo foi suficientemente pequeno para se ter uma boa idéia do tempo de execução.

5.2 A Metodologia

Para a avaliação da implementação utilizamos um gerador de grafos aleatórios, escrito por C. McGeoch e usado num *workshop* de implementação de algoritmos do centro DIMACS [Dim93]. Com ele geramos dois tipos de grafos (T1 e T2) com dois tamanhos cada (pequeno e grande). Os grafos de tipo T2 têm 2/3 mais arestas do que os de tipo T1, mas ambos são relativamente esparsos. Na tabela 1 apresentamos o número de vértices e arestas dos 2 tipos de grafo utilizados nas medições.

	Pequeno		**Grande**	
	Vértices	Arestas	Vértices	Arestas
T1	20.000	30.000	40.000	60.000
T2	20.000	50.000	40.000	100.000

Tabela 1: Tamanho dos grafos utilizados para as medições.

Para fornecer uma idéia de desempenho da implementação paralela em relação à uma implementação seqüencial, utilizamos uma implementação em FORTRAN do algoritmo de Micali e Vazirani feita por Mattingly e Richey [MR93].

As medicões do código tanto seqüencial quanto paralelo foram feitas sobre cinco instâncias de cada um dos dois tamanhos dos 2 tipos de grafo. Cada instância foi gerada com uma semente diferente para o gerador de grafos. Os tempos da implementação seqüencial são médias das cinco instâncias mencionadas acima.

No caso paralelo, para um determinado número de *threads*, cada instância foi resolvida 3 vezes, pois o tempo de execução apresenta uma certa variação entre uma execução e outra, mesmo com a mesma entrada. Tomou-se a média dessas 3 execuções, e o tempo paralelo para um certo tipo e tamanho de grafo é a média das médias de cada uma das 5 instâncias. Esse processo foi feito para 1, 2, 4 e 6 *threads*.

5.3 Os Resultados

Os resultados das medições da implementação paralela estão na tabela 2. Estes resultados são preliminares, em particular por causa das três seguintes observações. Em primeiro lugar,

os tempos aqui reportados são para uma implementação que não refaz as buscas quando inconsistências são detectadas (o vértice livre é simplesmente abandonado). Entretanto não esperamos mudanças significativas no tempo quando o algoritmo estiver implementado por completo, pois medições indicaram que nos grafos testados o número de buscas que teriam que ser refeitas é muito pequeno quando comparado ao total de buscas (em torno de 1%). Em segundo lugar, o código FORTRAN seqüencial não foi compilado com *flags* de otimização; quando isso for feito o tempo seqüencial deve diminuir de forma significativa. Em terceiro lugar, foram testados apenas 2 tipos de grafos, provenientes do mesmo gerador. Seria prematuro extrapolar os resultados aqui observados para outras classes de grafos.

	Seqüencial	1 CPU	2 CPU	4 CPU	6 CPU
Pequeno T2	4,87	(8,87 – 8,94)	(5,18 – 6,05)	(3,69 – 5,02)	(3,02 – 5,25)
Grande T2	10,86	(29,19 – 29,35)	(17,57 – 19,58)	(10,78 – 13,49)	(9,33 – 13,97)
Pequeno T1	13,17	(11,9 – 11,94)	(6,31 – 6,55)	(3,72 – 4,23)	(2,88 – 3,61)
Grande T1	37,48	(45.85 – 45,97)	(24.78 – 25.24)	(13,7 – 14.51)	(9.94 – 10.97)

Tabela 2: Tempos da implementação seqüencial e paralela.

Os melhores resultados obtidos estão na tabela 3.

	Pequeno		**Grande**	
	Seqüencial	Paralelo	Seqüencial	Paralelo
T2	4,87	(3,23 – 5,02)	10,86	(10,78 – 13,49)
T1	13,17	(2,88 – 3,61)	37,48	(9,94 – 10,97)

Tabela 3: Tabela dos melhores tempos.

O melhor desempenho da implementação paralela foi observado em grafos grandes do tipo T1 (mais esparsos). Na figura 6, observamos a sua aceleração com o aumento do número de processadores utilizados. Uma análise mais detalhada destes resultados, levando em conta diversos fatores tais como tempo gasto nas diversas fases do algortimo, atrasos causados pela memória compartilhada, etc, será parte da continuação deste trabalho.

6 Conclusões

As conclusões básicas a que chegamos até o momento são as seguintes.

O algoritmo de Micali e Vazirani possui restrições que dificultam a obtenção de uma implementação paralela. Dentre as restrições existentes destacamos o excesso de sincronização necessária e o aumento do número de tarefas devido ao paralelismo. Portanto não acreditamos que seja um caminho viável. A adaptação do algoritmo de Edmonds por outro lado é muito mais simples e com ela conseguimos resultados encorajadores. Em particular, considerando a dificuladade que os teóricos vêm encontrando para formular um algoritmo

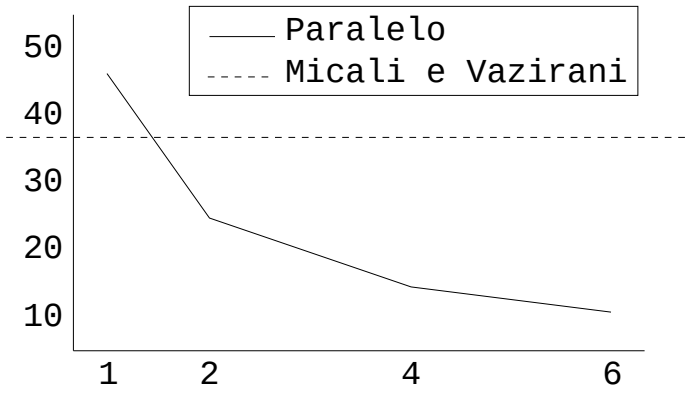

Figura 6: Gráfico de performance em grafos do tipo T1 grandes.

paralelo eficiente para o problema do emparelhamento, achamos que a obtenção de qualquer aceleração sobre um bom código seqüencial pode ser vista como um avanço.

Futuramente esperamos aperfeiçoar a implementação, otimizando algumas operações de baixo nível, testando-a em tipos mais variados de grafos, e analisando com cuidado os resultados experimentais obtidos.

Referências

[Ber57] Berge, C. Two theorems in graph theory. *Proc. Nat. Acad. Sci.*, 43: 842–844, 1957.

[Blu94] N. Blum. A new approach to maximum matching general graph. Technical Report 94-06-23, Univerität Bonn, Bonn, Germany, 1994.

[Dim93] David S. Johnson and Catherine C. McGeoch (editors). *First* DIMACS *Implementation Challenge — Network Flows and Matching.* American Mathematical Society, DIMACS Series in Mathematics and Theoretical Computer Science, volume 12, 1993.

[Edm65] J. Edmonds. Paths, trees, and flowers. *Canad. J. Math.*, 17: 449–467, 1965.

[EK75] S. Even and O. Kariv. An $O(n^{2.5})$ algorithm for maximum matching in general graphs. *Proc. 16th Annual IEEE Symposium on Foundations of Computer Science*, 100–112, 1975.

[Gab76] H. N. Gabow. An efficient implementation of Edmonds' maximum matching algorithm. *J. Assoc. Comput. Mach.*, 23: 221–234, 1976.

[HK73] J. E. Hopcroft e R. M. Karp. An $n^{5/2}$ algorithm for maximum matching in bipartite graphs. *SIAM J. Comput.*, 2: 225–231, 1973.

[MR93] R. B. Mattingly and N. P. Richey. Serial and parallel algorithms for cardinality matching problems. In David S. Johnson and Catherine C. McGeoch, editors, *Network Flows and Matching* – 1^{st} DIMACS *Implementation Challenge*, 539–556. American Mathematical Society, 1993.

[MV80] S. Micali and V. V. Vazirani. An $O(\sqrt{|V|}|E|)$ algorithm for finding maximum matching in general graphs. In *Proceedings of the IEEE* 21*st Annual Symposium on Foundations of Computer Science*, 17–27, 1980.

[MVV87] K. Mulmuley, U. V. Vazirani, and V. V. Vazirani. Matching is as easy as matrix inversion. *Combinatorica*, 7: 105–113, 1987.

[NR59] R. Z. Norman and M. O. Rabin. An algorithm for a minimum cover of a graph *Proc. Amer. Math. Soc.*, 10: 315–319, 1959.

[PL88] P. A. Peterson and Michael C. Loui. The general maximum matching algorithm of Micali and Vazirani. *Algorithmica*, 3: 511–533, 1988.

Upper Bounds for Minimum Covering Codes by Tabu Search

Walter A. Carnielli
Depto. de Filosofia, IFCH, UNICAMP
Caixa Postal 6110
13083-970 Campinas, SP
walter@turing.unicamp.br

Emerson L. do Monte Carmelo
Depto. de Matemática, IMECC, UNICAMP
Caixa Postal 6055
13083-970 Campinas, SP

Marcus V. S. Poggi de Aragão
Cid C. de Souza
Depto. de Ciência da Computação, IMECC, UNICAMP
Caixa Postal 6176
13083-970 Campinas, SP
{cid,poggi}@dcc.unicamp.br

Abstract: Let V_k^n denote the set of all n-dimensional vectors whose components are integers between 0 and $k-1$. A subset of V_k^n is called a *code*. A code C is said to be an R-covering of V_k^n if, for all $y \in V_k^n$, there exists a vector $x \in C$ within Hamming distance R from y. The *covering* problem is defined as the problem of finding a covering code of minimum size. In this paper we formulate this problem as a dominating set problem in a graph and propose a Tabu Search algorithm to solve it. The computational results obtained by our approach are also discussed. Though we have not been able to produce any new upper bounds, many of our bounds correspond to the best known upper bounds of the literature.

Keywords: code covering problems, upper bounds, dominating sets in graphs, tabu search.

1 Introduction

Consider the two problems below:

Problem 1: Suppose that to enter at the University a student has to make an Exam consisting of n multiple choice questions with q choices per question. To pass the exam, the student can give at most R wrong questions. If the student is not limited to give one set of n answers, what is the minimum number of sets of n answers he (she) should give to ensure the entrance at the University ?

Problem 2: In a chessboard, how many rooks does one need to ensure that each position of the board is reached by at least one rook with a single movement ? More generally, in a n-dimensional chessboard with k^n positions, how many hyper-rooks does one need to ensure that each position of the board is reached by at least one rook with at most R movements ?

These two problems are examples of the code covering problem that we now describe more formally.

Let V_k^n denote the set of all n-dimensional vectors whose components are integers between 0 and $k-1$. Given two vectors x and y in V_k^n, the **Hamming distance** between x and y, denoted by $d(x,y)$, is the number of components in which x and y differ. Any subset of V_k^n is called a **code**. A code C is said to be an *R-covering of V_k^n* if, for all $y \in V_k^n$, there exists a vector $x \in C$ such that $d(x,y) \leq R$. The **code covering problem** consists of finding a covering code of minimum size. For a given triple (n,k,R) the size of a minimum code is denoted by $\gamma(n,k,n-R)$. For $R=1$, the notation $\sigma(n,k)$ is also used.

A special instance of the problem is when $k=3$ and $R=1$. In this case, the problem is called the **football pool problem** since for each component of a vector in V_3^n, like in each match of a football pool coupon, there are three possible outcomes. In most countries, there is a prize for the players who missed at most one of the results. So, the natural question is how many coupons one has to bet to be sure to win a prize.

The code covering problem has been extensively studied in the literature (c.f., [1], [2]). We refer to to the works of Monte Carmelo ([8]) and Östergard ([9]) for surveys on the main results. Few exact values of $\gamma(n,k,s)$ are known and most of the effort are concentrated in improving the existing lower and upper bounds. Improving lower bounds is a hard task and different techniques have been used to generate nontrivial lower bounds. One way to improve an upper bound is to exhibit a covering code with size smaller than the best known one. Several of the known upper bounds have been generated following this constructive approach. In this paper we formulate the code covering problem as a **minimum dominating set problem** in a graph and propose a Tabu-Search algorithm to construct small dominating sets.

The paper is divided as follows. In Section 2 we recall the definition of a dominating set in a graph and show how the code covering problem reduces to an instance of such problem. In Section 3 we present a Tabu Search algorithm for the dominating set problem and we give some implementation details. Section 4 describes our computational experiments, compares our results with those found in the literature and draws some conclusions.

2 Covering Codes and Dominating Sets in Graphs

In this section we define the dominating set problem in graphs and show that the code covering problem can be reduced to the former.

Let $G=(V,E)$ be an undirected graph with node set V and edge set E. The node set $U \subseteq V$ is said to be a dominating set of G if, for every node v in V, either v is in U or there exists a node u in U such that u is adjacent to v (*i.e.*, $(u,v) \in E$). The dominating set problem is the problem of finding a dominating set in the graph whose size is minimum. This problem is *NP*-hard.

Given the set V_k^n of all n-dimensional vectors with components in $\{0,1,\ldots,k-1\}$ and an integer R with $0 \leq R \leq n$, let us construct the graph $G(n,k,R)=(V,E)$ as follows. For each vector x in V_k^n we associate a node x in V. The edge $e=(x,y)$ is in E if and only if the vectors x and y satisfy $0 < d(x,y) \leq R$. Thus, we have that $|V| = k^n$ and that

$G(n,k,R)$ is a regular graph where the degree of every node is given by

$$\alpha(n,k,R) = \sum_{i=1}^{R} \left(\begin{array}{c} n \\ i \end{array} \right) (k-1)^i.$$

The construction above is illustrated by the following example. Consider the set $V_3^2 = \{00, 01, 02, 10, 11, 12, 20, 21, 22\}$ and let $R = 1$. The graph G is shown in Figure 1.

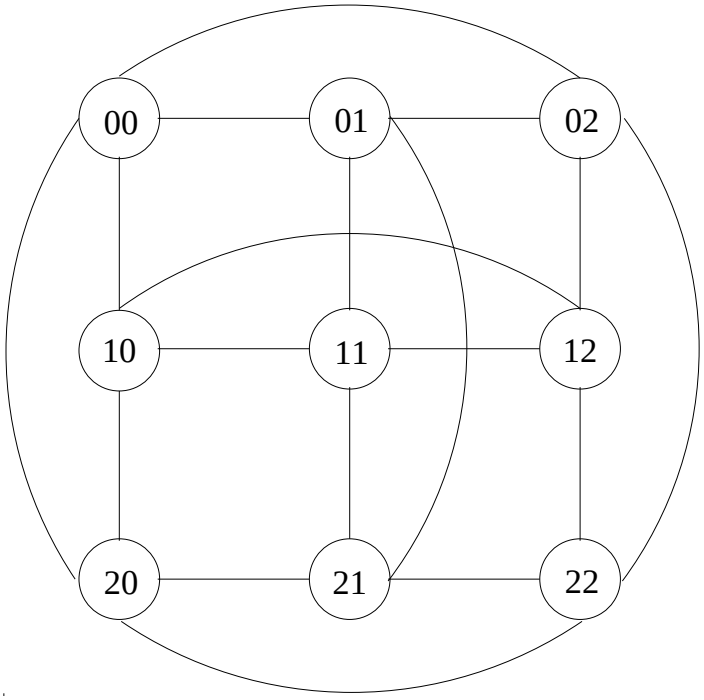

Figure 1: $G(2,3,1)$

It is very easy to see that the dominating sets in $G(n,k,R)$ are in one-to-one correspondence with the R-covering codes in V_k^n. Thus, a minimum R-covering code in V_k^n can be found by solving the minimum dominating set problem in $G(n,k,R)$.

A well known lower bound on the cardinality of a covering code is the so called *sphere covering bound* ([9]) given by:

$$\gamma(n,k,n-R) \geq \frac{k^n}{\alpha(n,k,R)+1}$$

If we formulate the dominating set problem on $G(n,k,R)$ as an integer program, it corresponds to a set covering problem. Each node in $G(n,k,R)$ defines a constraint which implies that the sum of the variable corresponding to this node and those corresponding to all its adjacent nodes is at least 1. Thus there are $\alpha(n,k,R)+1$ variables in each constraint. It turns out that the optimal value of the linear relaxation is equal to the sphere bound. To check that, assume that $\frac{1}{\alpha(n,k,R)+1}$ is assigned to each variable. This solution satisfies

all constraints at equality. The corresponding objective function value is then the sphere covering bound. Notice now that this nonbasic solution is optimal for the linear relaxation. To verify its optimality, just observe that the corresponding linear program is symmetric and therefore an analogous solution with this same objective value is feasible for the dual.

3 A Tabu Search Algorithm

In Section 2 we have reduced the code covering problem to a dominating set problem in a graph. We are interested here in finding good upper bounds for the covering problem and one way to do it is to have an algorithm that constructs small dominating sets for the graph described in Section 2. To reach our goal we have designed a Tabu Search (TS) algorithm for the dominating set problem.

Tabu Search is an adaptive procedure that have been applied to a wide range of combinatorial optimization problems ([3] and [4]). Like other local search algorithms, the procedure starts with an initial solution and explores the solution space by iteratively changing from the current solution to another solution via a well-defined perturbation process. The set of all solutions reachable from a given solution s via the perturbation process defines the *neighborhood* of s. The criterion to select the next solution to be visited uses a greedy strategy.

The set of solutions visited by TS forms a path in the solution space. The main goal of TS is to avoid the existence of cycles in this path while inducing a broad search of the solution space as well as an intensive search on promising regions. To move from the current solution to another one, the *neighborhood* of a solution has to be defined. However we can only move from the current solution to one of the solutions in its neighborhood when the corresponding movement is not Tabu, *i.e.*, the movement is not forbidden. We describe below the details of our algorithm to find dominating sets in the graph $G(n, k, R) = (V, E)$.

The solution space used by our TS algorithm for the dominating set problem is the set of all subsets of nodes in V. Notice that this choice implies that the algorithm may end up with a solution which is not a dominating set of $G(n, k, R)$. The goal in enlarging the solution space is to have more freedom to escape from local minima of the objective function. This technique has been also used successfully in other applications of local search algorithms for hard combinatorial optimization problems (*cf.*, [5]). Below we describe how dominating sets are generated in this space.

The neighborhood we have defined is quite natural. If $U \subseteq V$ is the current solution, the neighborhood of U is the set of all solutions obtained by U either by removing one of its nodes or by including a node of $V \setminus U$ to U. We still have to guarantee that nondominating subsets of V are discarded and to define the tabu movements.

To avoid ending up with a solution that is not a dominating set, we have adapted the (objective) function to evaluate the subsets of V in the solution space. Since we are looking for a minimum-size dominating set, the natural objective function is the size of the current solution. However, this clearly would lead to nondominating sets (since we have enlarged the solution space). Therefore, we have modified the objective function to include a second term which penalizes the sets that are not dominating. Thus the objective function is to

minimize the sum the following terms: the size of the set and the product of a penalty factor and the number of nodes in V not covered by the current solution. A sufficiently large penalty factor ensures that the procedure will converge to a dominating set. We now present the general framework of the algorithm.

Procedure Tabu-Search

Step 1 Initializations.

- *Construct an initial cover U and let U_i correspond to the subset of V_k^n obtained from U by switching the i^{th} element. Then the neighborhood of U is given by $N(U) = \{U_1, U_2, \ldots, U_n\}$.*
- *Set all movements as nontabu, i.e. $T \longleftarrow \emptyset$.*
- *Since U is a cover, set it as the best known cover, i.e. $U^* \longleftarrow U$.*
- *Set $f(U) = |U|$.*

Step 2 Main Loop. *Repeat* NUMBER-OF-TRIALS *times. After that, repeat only if the solution was improved in last trial.*

- *Set penalty change strategy, i.e. define function $P(q)$ which stands for the penalty associated to each uncovered node.*

  for q **from** *1* **to** NUMBER-OF-PENALTY-LEVELS

  - *Compute function value with penalty $P(q)$;*

    for k **from** *1* **to** NUMBER-OF-ITERATIONS

    - *Let i^* minimize $f(U_i)$ for $U_i \in N(U)$. If $i \in T$, then U_i is considered only if an* aspiration criterion *is met, in this case this stands for finding a new best cover.*
    - *If $f(U_{i^*}) \geq f(U)$, add i^* to the tabu list, $T \longleftarrow T \bigcup \{i^*\}$, and compute an iteration to remove it from T.*
    - *Update current solution, i.e. $U \longleftarrow U_{i^*}$ and $f(U) \longleftarrow f(U_{i^*})$.*
    - *Update tabu list, i.e. take from T the tabu nodes due to removal.*
    - *If U is a cover, check if a new best cover was found, i.e. if $|U| < |U^*|$. If so, $U^* \longleftarrow U$.*

    endfor k

  endfor q

Although the ideas behind our Tabu-Search procedure where discussed above, several issues still unanswered. The first one refers to the *penalty change strategy* which appears in the framework as function $P(q)$. To derive some intuition on how should $P(q)$ behave we ran some preliminary tests. One first conclusion was that increasing $P(q)$ slowly until a large value was met and then reducing $P(q)$ to zero at once was best. Other strategies such as doing the opposite, decreasing slowly and going to a large value at once, and increasing and decreasing slowly produced poor results. One second conclusion addresses what is a large value for $P(q)$. In fact, large here is anything greater than one. We obtained our best results by letting $P(q)$ grow from 0 to 20 in 20 steps, *i.e.* using NUMBER-OF-PENALTY-LEVELS equal to 20.

The number of iterations a switch remains tabu, i.e. the tabu tenure value, is given by $(\beta + \delta.\text{unif}(0,1)).a$ where β and δ took values around 10, and where a assume value 1 when the objective function was increased when the node enters the current solution and 2 when it leaves. In other words, we used a randomized asymmetric tabu tenure, having a as asymmetry factor. This asymmetry is reasonable since there are a lot more node outside a "small" cover than in the cover itself. Moreover, this seems to help in breaking the graph regularity.

The NUMBER-OF-ITERATIONS is the last parameter left to define. We used values between half the number of nodes in the graph and two times this number. Changes were made to put more or less effort, *i.e.* cpu time, on the search.

4 Results and Conclusions

We applied our Tabu-Search procedure to 11 well studied problems in the literature. The results obtained are summarized in Table 1 below where SB stands for the *Sphere Bound*, LB and UB for the best to date known lower and upper bounds respectively, and TS for the value we obtained with our procedure.

It is worth noting that for $\gamma(6,3,1)$ we were able to obtain codes of cardinality 73 within less than 15 minutes on a SUN SPARC 1000. This bound was obtained for the first time in [7] using the simulated annealing algorithm proposed in [11] (in which a covering code of 74is described). The authors do not give the CPU time necessary to achieve the 73 bound. It is mentioned however that the Simulated Annealing algorithm with a very small cooling rate (large CPU time) has not been able to produce a code of size 72. For $\gamma(7,3,1)$ an observation is made that 50 runs of 2 hours of CPU time of a VAX 11/785 were necessary to compute the bound of 186. Two of the fifty runs have reached this bound. In [6] no comments are addressed to the amount of time necessary to obtain the upper bounds. The same observation remains valid for [10].

It can be seen that there are very few information concerning the time spent in trying to generate good upper bounds for the function γ. Thus, from this point of vue, any attempt to compare the different algorithms is a very hard task. It remains to compare the upper bounds they produce.

We notice that many of the known upper bounds shown in Table 1 have been obtained after carefully tuning of the parameters of the algorithm for a particular instance of interest. The parameters of our TS algorithm have been set in order to produce good solutions for many different triples (n,k,R) and not for just one triple of these values. Despite of that, in 8 out of 11 tests shown in Table 1 we were able to reproduce the best known upper bounds. In two of the three cases where we could not achieve the best bound, our solution is only one unity larger. The only instance for which we failed to get very close (or equalize) the best bound was the $(7,3,1)$ instance. The result given in Table 1 corresponds to a single run of approximately 6 hours of CPU time.

It should be mentioned that the results given in this paper come from our very first implementation of the TS algorithm for the minimum covering code problem. Therefore, they are preliminary results and, in this context, they can be considered as very encouraging. We believe that there is room for further improvements in the bounds generated by TS but, for this, a more rigorous choice of the set of parameters of TS has to be done.

n	k	R	$\|V_k^n\|$	$\alpha(n,k,R)$	SB	LB	UB	TS
4	5	1	625	16	36.76	45	51	52
4	5	2	625	112	5.53	9	11	11
5	3	2	243	50	4.76	6	8	8
5	4	2	1024	105	9.66	12	16	16
6	3	1	729	12	56.08	63	73	73
6	3	2	729	72	9.99	12	17	17
6	4	2	4096	153	26.60	28	64	65
6	4	3	4096	693	5.90	8	16	16
7	3	1	2187	14	145.80	147	186	192
7	3	2	2187	98	22.09	26	34	34
7	3	3	2187	388	5.62	7	12	12

Table 1

References

[1] W. A. Carnielli, On Covering and Coloring Problems for Rook Domains, Discrete Mathematics **57** (1985), 9-16.

[2] W. A. Carnielli, Hyper-rook domain inequalities, Studies in Applied Mathematics **82** (1990), 50-69

[3] F. Glover, Tabu Search - Part I, ORSA Journal on Computing **1** (1989), 190-206.

[4] F. Glover, Tabu Search - Part II, ORSA Journal on Computing **2** (1990), 4-32.

[5] D.S. Johnson, C.R. Aragon, L.A. McGeoch, and C. Schevon (1989). Optimization by Simulated Annealing: An Experimental Evaluation: Part I, Graph Partitioning, *Operations Research* **37**, 865-892.

[6] K. U. Koschnick (1993), A new Upper Bounds for the Football Pool Problem for nine matches, Journal of Combinatorial Theory, Series A, **62**, 162-167.

[7] P. J. M. van Laarhoven, E. H. L. Aarts and J. H. van Lint (1989), New Upper Bounds for the Football Pool Problem for 6, 7 and 8 matches, Journal of Combinatorial Theory, Series A, **52**, 304-312.

[8] E. L. do Monte Carmelo (1995), O Problema das Hipertorres e Partições Polarizadas Finitas e Infinitas, M.Sc. Dissertation, Department pf Mathematics, Institut of Mathematics, Statistics and Computer Science, Universidade Estadual de Campinas, Brazil.

[9] P. R. J. Östergard, Construction Methods for Covering Codes, Department of Computer Science (Digital Systems Laboratory), Helsinky University of Technology, Espoo, Finland.

[10] P. R. J. Östergard (1995), New Upper Bounds for the Football Pool Problem for 11 and 12 matches, to appear in Journal of Combinatorial Theory, Series A.

[11] L. T. Wille (1987), The football pool problem for 6 matches: a new upper bound obtained by simulated annealing, Journal of Combinatorial Theory, Series A, **45**, 171-177.

Relatórios Técnicos – 1995

95-01 **Paradigmas de algoritmos na solução de problemas de busca multidimensional,** *Pedro J. de Rezende, Renato Fileto*

95-02 **Adaptive enumeration of implicit surfaces with affine arithmetic,** *Luiz Henrique de Figueiredo, Jorge Stolfi*

95-03 **W3 no Ensino de Graduação?,** *Hans Liesenberg*

95-04 **A greedy method for edge-colouring odd maximum degree doubly chordal graphs,** *Celina M. H. de Figueiredo, João Meidanis, Célia Picinin de Mello*

95-05 **Protocols for Maintaining Consistency of Replicated Data,** *Ricardo Anido, N. C. Mendonça*

95-06 **Guaranteeing Full Fault Coverage for UIO-Based Methods,** *Ricardo Anido and Ana Cavalli*

95-07 **Xchart-Based Complex Dialogue Development,** *Fábio Nogueira de Lucena, Hans K.E. Liesenberg*

95-08 **A Direct Manipulation User Interface for Querying Geographic Databases,** *Juliano Lopes de Oliveira, Claudia Bauzer Medeiros*

95-09 **Bases for the Matching Lattice of Matching Covered Graphs,** *Cláudio L. Lucchesi, Marcelo H. Carvalho*

95-10 **A Highly Reconfigurable Neighborhood Image Processor based on Functional Programming,** *Neucimar J. Leite, Marcelo A. de Barros*

95-11 **Processador de Vizinhança para Filtragem Morfológica,** *Ilka Marinho Barros, Roberto de Alencar Lotufo, Neucimar Jerônimo Leite*

95-12 **Modelos Computacionais para Processamento Digital de Imagens em Arquiteturas Paralelas,** *Neucimar Jerônimo Leite*

95-13 **Modelos de Computação Paralela e Projeto de Algoritmos,** *Ronaldo Parente de Menezes e João Carlos Setubal*

95-14 **Vertex Splitting and Tension-Free Layout,** *P. Eades, C. F. X. de Mendonça N.*

95-15 **NP-Hardness Results for Tension-Free Layout,** *C. F. X. de Mendonça N., P. Eades, C. L. Lucchesi, J. Meidanis*

95-16 **Agentes Replicantes e Algoritmos de Eco,** *Marcos J. C. Euzébio*

Departamento de Ciência da Computação — IMECC
Caixa Postal 6176
Universidade Estadual de Campinas
13081-970 – Campinas – SP
BRASIL

reltec@dcc.unicamp.br